Titre: Title:	Exploiting Global Constraints for Search and Propagation
Auteur: Author:	Alessandro Zanarini
Date:	2010
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Zanarini, A. (2010). Exploiting Global Constraints for Search and Propagation [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/299/

UNIVERSITÉ DE MONTRÉAL

EXPLOITING GLOBAL CONSTRAINTS FOR SEARCH AND PROPAGATION

ALESSANDRO ZANARINI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

EXPLOITING GLOBAL CONSTRAINTS FOR SEARCH AND PROPAGATION

présentée par: ZANARINI Alessandro
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury constitué de:

M. GALINIER, Philippe, Doct., président.
M. PESANT, Gilles, Ph.D., membre et directeur de recherche.
M. ROUSSEAU, Louis-Martin, Ph.D., membre.
M. VAN BEEK, Peter, Ph.D., membre externe.

To my family

Abstract

This thesis focuses on Constraint Programming (CP), that is an emergent paradigm to solve complex combinatorial optimization problems. The main contributions revolve around constraint filtering and search that are two main components of CP. On one side, constraint filtering allows to reduce the size of the search space, on the other, search defines how this space will be explored. Advances on these topics are crucial to broaden the applicability of CP to real-life problems.

For what concerns constraint filtering, the contribution is twofold: we firstly propose an improvement on an existing algorithm of the relaxed version of a constraint that frequently appears in assignment problems (`soft_gcc`). The algorithm proposed outperforms the previously known in terms of time-complexity both for the consistency check and for the filtering and in term of ease of implementiation. Secondly, we introduce a new constraint (both hard and soft version) and associated filtering algorithms for a recurrent sub-structure that occurs in assignment problems with heterogeneous resources (`hierarchical_gcc`). We show promising results when compared to an equivalent decomposition based on `gcc`.

For what concerns search, we introduce algorithms to count the number of solutions for two important families of constraints: occurrence counting constraints, such as `alldifferent`, `symmetric_alldifferent` and `gcc`, and sequencing constraints, such as `regular`. These algorithms are the building blocks of a new family of search heuristics, called constraint-centered counting-based heuristics. They extract information about the number of solutions the individual constraints admit, to guide search towards parts of the search space that are likely to contain a high number of solutions. Experimental results on eight different problems show an impressive performance compared to other generic state-of-the-art heuristics.

Finally, we experiment on an already known strong form of constraint filtering that is heuristically guided by the search (quick shaving). This technique gives mixed results when applied blindly to any problem. We introduced a simple yet very effective estimator to dynamically disable quick shaving and showed experimentally very promising results.

Résumé

Cette thèse se concentre sur la Programmation par contraintes (PPC), qui est un paradigme émergent pour résoudre des problèmes complexes d'optimisation combinatoire. Les principales contributions tournent autour du filtrage des contraintes et de la recherche; les deux sont des composantes clé dans la résolution de problèmes complexes à travers la PPC. D'un côté, le filtrage des contraintes permet de réduire la taille de l'espace de recherche, d'autre part, la recherche définit la manière dont cet espace sera exploré. Les progrès sur ces sujets sont essentiels pour élargir l'applicabilité de CP à des problèmes réels.

En ce qui concerne le filtrage des contraintes, les contributions sont les suivantes: premièrement, on propose une amélioration sur un algorithme existant de la version relaxée d'une contrainte commune qui apparaît souvent dans les problèmes d'affectation (`soft_gcc`). L'algorithme proposé améliore en termes de complexité soit pour la cohérence, soit pour le filtrage et en termes de facilité d'implémentation. Deuxièmement, on introduit une nouvelle contrainte (soit dure soit relaxée) et les algorithmes de filtrage pour une sous-structure récurrente qui se produit dans les problèmes d'affectation des ressources hétérogènes (`hierarchical_gcc`). Nous montrons des résultats encourageants par rapport à une décomposition équivalente basée sur `gcc`.

En ce qui concerne la recherche, nous présentons tout d'abord les algorithmes pour compter le nombre de solutions pour deux importantes familles de contraintes: les contraintes sur les occurrences, par exemple, `alldifferent`, `symmetric_alldifferent` et `gcc`, et les contraintes de séquence admissible, telles que `regular`. Ces algorithmes sont à la base d'une nouvelle famille d'heuristiques de recherche, centrées sur les contraintes et basées sur le dénombrement. Ces heuristiques extraient des informations sur le nombre de solutions des contraintes, pour guider la recherche vers des parties de l'espace de recherche qui contiennent probablement un grand nombre de solutions. Les résultats expérimentaux sur huit différents problèmes montrent une performance impressionnante par rapport à l'état de l'art des heuristiques génériques.

Enfin, nous expérimentons une forme forte, déjà connue, de filtrage qui est guidée par la recherche (quick shaving). Cette technique donne des résultats soit encourageants soit mauvais lorsqu'elle est appliquée aveuglément à tous les problèmes. Nous avons introduit un estimateur simple mais très efficace pour activer ou désactiver dynamiquement le quick shaving; de tests expérimentaux ont montré des résultats très prometteurs.

Condensé en Français

Les Problèmes de Satisfaction de Contraintes ou CSP (Constraint Satisfaction Problem) sont des problèmes mathématiques où on doit trouver des états ou des objets dans un système qui satisfont un certain nombre de contraintes ou de critères. Les contraintes ont émergé comme un domaine de recherche qui regroupe des chercheurs à partir d'un certain nombre de domaines, y compris l'intelligence artificielle, les langages de programmation et la programmation logique. Les réseaux de contraintes et les problèmes de satisfaction de contraintes ont été étudiés depuis les années soixante-dix.

Le domaine de la Programmation par Contraintes (PPC) est née de langages de programmation et de la Programmation Logique dans les années 1980 et est aujourd'hui un paradigme efficace pour résoudre les problèmes d'optimisation combinatoire. La PPC a été appliquée avec succès à de nombreux domaines; on mentionne le traitement de la langue naturelle, les systèmes de base de données, la biologie moléculaire, les transports, la logistique, la chaîne d'approvisionnement et les problèmes de stockage, gestion du personnel, la planification des ressources, la conception et vérification de systèmes embarqués.

Les contraintes globales constituent un aspect clé de la PPC, car elles capturent des sous-structures récurrentes dans les problèmes : elles facilitent le processus de modélisation pour l'utilisateur et plus important encore, elles ont permis une amélioration majeure du processus de solution en évitant des calculs inutiles lors de la recherche de solutions cohérentes au problème. Offrir de nouvelles contraintes globales ou améliorer les méthodes de filtrage de contraintes déjà connues ont donc un impact direct sur les capacités et l'efficacité que la PPC offre à l'utilisateur.

Cependant, le filtrage souvent ne suffit pas pour résoudre les problèmes de la vie réelle, donc la recherche est une composante nécessaire pour évaluer différentes voies qui peuvent conduire aux solutions d'un problème. Différentes heuristiques de recherche génériques ont été développés au fil des ans, cependant l'utilisateur est toujours confronté au choix entre dépenser des ressources et du temps pour développer des heuristiques très efficaces mais spécifiques au problème en question ou utiliser des heuristiques de recherche générique avec le risque d'obtenir une performance médiocre. Sur ce front d'autres paradigmes (la Programmation Linéaire en Nombre Entiers par exemple) offrent à l'utilisateur des heuristiques efficaces et intégrées au solveur qui peuvent être utilisées directement; à cet égard, la PPC est en retard et elle n'a toujours pas fourni des heuristiques de recherche entièrement automatisées, génériques et aussi efficaces. Chaque avancement dans cette direction est essentiel pour élargir l'utilisation de la PPC par les professionnels.

Cette thèse s'articule autour de ces deux sujets: les algorithmes de filtrage pour les contraintes globales et les heuristiques de recherche. D'un côté on améliore les algorithmes de filtrage existants et on propose une nouvelle contrainte globale; dans l'autre, on introduit une manière complètement nouvelle de concevoir des heuristiques de recherche qui exploite d'une façon mieux intégrée les contraintes globales du modèle en comptant le nombre de solutions de chaque contrainte individuelle. Cette nouvelle famille d'heuristiques vise à être générique et pourtant efficace pour résoudre les problèmes de la vie réelle. Enfin, on touche un sujet qui se trouve entre les deux précédents, qui est une forme forte de cohérence qui est heuristiquement guidée par la recherche (Quick Shaving).

Plus précisément, les principales contributions sont les suivantes:

- un nouvel algorithme de filtrage amélioré pour l'une des contraintes les plus communes, qui est la contrainte globale de cardinalité en version molle (`soft_gcc`) (paru dans [111])
- une nouvelle contrainte globale qui est une généralisation de la contrainte globale de cardinalité et qui intègre la notion de ressources hétérogènes pour des problèmes d'affectation (paru dans [112])
- algorithmes de dénombrement pour la contrainte `alldifferent`, la version symétrique `symétrique_alldifferent`, la contrainte globale de cardinalité `gcc` et la contrainte `regular` (partiellement parus dans [113], [114] et [116])
- heuristiques basées sur le dénombrement qui extraient des informations sur le nombre de solutions des contraintes du modèle et orientent la recherche vers des parties de l'arbre de recherche qui sont fort probable de contenir un grand nombre de solutions (partiellement parus [113], [114])
- une technique de Quick Shaving amélioré qui est une forme forte de filtrage guidé par la recherche (paru dans [115])

Nous allons décrire les contributions avec plus des détails dans les sections suivantes.

Contrainte globale de cardinalité en version molle

De nombreux problèmes de la vie réelle sont sur-contraints puisque la restriction et le nombre élevé de contraintes peut rendre les problèmes irréalisables. Dans ces situations, il convient de trouver une solution qui viole partiellement certaines contraintes, mais qui est toujours intéressante pour l'utilisateur. Les contraintes peuvent être regroupées en *contraintes dures* qui ne peuvent pas être violées, et *contraintes souples ou molles* qui peuvent être (partiellement) violées. Typiquement, les contraintes dures sont utilisées pour la modélisation de la structure inhérente du problème et les contraintes liées aux préférences que l'utilisateur

souhaite introduire dans le modèle sont définies comme molles.

Pour chaque contrainte molle, une variable représente le coût de la violation et une fonction associée mesure la violation de la contrainte. L'objectif principal est alors de trouver une solution qui minimise la violation totale qui est habituellement une fonction de violations des contraintes individuelles; des exemples courants sont la somme de toutes les violations ou le maximum.

Nous avons travaillé sur la version molle de sous structure le plus commun dans le problème combinatoire qui est la contrainte globale de cardinalité - `gcc`. Cette contrainte limite un ensemble de variables en spécifiant le nombre minimum et le nombre maximum d'occurrences de chaque valeur dans une solution.

Les algorithmes connus de filtrage de la contrainte `soft_gcc` utilisent la théorie des graphes, en particulier l'algorithme de flot maximum à coût minimum.

Nous proposons un nouvel algorithme de filtrage qui permet l'utilisation de la théorie des couplages sur le graphe. L'idée principale est de calculer une solution partielle qui minimise la violation sur le nombre minimum d'occurrences et une autre en minimisant la violation sur le nombre maximum d'occurrences. Nous montrons en suite qu'il est possible de combiner les deux solutions partielles afin d'avoir une affectation complète avec une violation totale minimale.

Le nouvel algorithme a dans le pire cas une complexité en temps $O(\sqrt{n}m)$ pour vérifier la cohérence de la contrainte et de $O(m+n)$ pour filtrer les valeurs incohérentes (où n est la cardinalité de l'ensemble des variables et $m = \sum_i |D_i|$). Il est meilleur que les algorithmes précédents qui ont une complexité en temps de $O(n(m + n \log n))$ pour la cohérence et de $O(\Delta(m + n \log n))$ pour le filtrage (où $\Delta = min(n, k)$ et k est la cardinalité de l'union des domaines des variables).

Cette nouvelle solution a un impact direct sur la résolution des problèmes de la vie réelle puisque le filtrage plus efficace se traduit en la possibilité d'adresser des problèmes plus difficiles ou plus complexes. La contribution va au-delà de l'efficacité car elle touche des aspects d'ingénierie de logiciel et de difficulté d'implémentation. Du point de vue de l'implémentation, l'algorithme proposé suit de très près l'algorithme de filtrage du `gcc` donc les efforts pour le programmeur sont sensiblement réduits. Notre algorithme est déjà présent dans un solveur commercial, Comet ^{TM}par Dynadec, et utilisé pour résoudre des applications réelles.

Généralizations de la Contrainte Globale de Cardinalité pour Resources Hiérarchiques

Les problèmes d'allocation des ressources se présentent dans des problèmes de la vie réelle chaque fois qu'il est nécessaire d'affecter des ressources à des tâches qui doivent être achevées. Il peut être considéré comme une relation d'un à un ou, plus généralement, une relation de plusieurs à un dans laquelle les tâches peuvent être affectées à une ou plusieurs ressources. En général, pour chaque tâche un nombre minimum et maximum des ressources nécessaires sont définies. Les ressources peuvent être *homogènes* dans le sens où elles ont des capacités ou compétences identiques. Dans la programmation par contraintes, les problèmes d'allocation de ressources homogènes peuvent être facilement modélisés par une contrainte globale de cardinalité [89] (`gcc`), dans laquelle chaque ressource est représentée par une variable dont le domaine est l'ensemble des tâches; chaque tâche définit ses besoins en ressources au moyen des limites inferieures et supérieures sur le nombre d'occurrences. Cependant, pour certains problèmes du monde réel, ce scénario est trop simpliste: les ressources sont hétérogènes et les tâches exigent des ressources avec des capacités ou des niveaux de compétence différents.

Nous avons travaillé sur une contrainte globale qui est une généralisation de la contrainte globale de cardinalité et qui introduit la notion de ressources hétérogènes et des niveaux de compétence. En particulier, une ressource d'un niveau de compétence donné est capable de satisfaire les exigences des tâches de niveaux égaux ou inférieurs. Cette sous-structure peut être modélisée par un ensemble de contraintes globales de cardinalité, mais cela ne garantit pas la cohérence de domaine. La nouvelle contrainte globale que nous proposons et l'algorithme de filtrage associé basé sur la théorie des flots, est capable d'atteindre la cohérence de domaine. Les résultats expérimentaux montrent les avantages de notre algorithme par rapport à la modélisation à travers un ensemble de contraintes globales de cardinalité. Nous proposons également une version soft de la contrainte. Cette contribution affecte directement l'efficacité dans la résolution de problèmes contenant des aspects liés à l'affectation des ressources hétérogènes en donnant la possibilité de résoudre des problèmes plus gros ou bien plus complexes.

Algorithmes de dénombrement

Avec cette contribution, on a l'intention d'enrichir l'interface des contraintes (globales ou non) : les contraintes auront non seulement des méthodes pour assurer la cohérence et pour effectuer le filtrage des domaines, mais aussi des informations sur le nombre de solutions. En particulier, on propose d'extraire à partir de contraintes globales soit le nombre total de solutions ou soit, pour chaque paire de variable-valeur, la densité des solutions qui représente

combien de fois une certaine affectation fait partie d'une solution de la contrainte.

Tout d'abord, on a étudié les algorithmes pour compter le nombre de solutions pour la contrainte `alldifferent`. Ce problème est équivalent à calculer le permanent de la matrice d'adjacence du graphe représentant la contrainte ("value graph"). Le problème du calcul du permanent a été largement étudié dans le passé et il a été prouvé $\#P$-complète (sous des hypothèses raisonnables sur la complexité des algorithmes, il faut un temps exponentiel pour le calculer). On a donc exploré des algorithmes d'approximation; en particulier on a proposé d'ajouter la propagation à un algorithme d'approximation existant qui échantillonne aléatoirement des solutions. Le nouvel algorithme permet d'éviter le problème du rejet d'échantillon et il améliore significativement la qualité de l'approximation par rapport à l'algorithme original. On a également proposé d'utiliser des bornes supérieures connues pour calculer efficacement les densités des solutions. Cette dernière approche s'est révélée être le meilleur compromis entre la précision de l'approximation et l'efficacité (temps pour calculer l'approximation). Les résultats expérimentaux montrent qu'on peut s'attendre à des approximations aussi bonnes que l'algorithme basé sur l'échantillonnage mais en prenant un centième du temps. L'approche a été étendue aussi à les contraintes `symmetric_alldifferent` et `gcc`.

On a ensuite exploré des algorithmes de dénombrement pour la contrainte `regular`. Notez que cette contrainte est assez générale et permet de modéliser parmi les autres aussi les contraintes de séquence `sequence`, de patrons `pattern` et `stretch`. Essentiellement, on exploite la même structure de données créée par l'algorithme de filtrage (un graphe à couches). Chaque chemin du graphe à partir de la source et se terminant dans le puits représente une solution de la contrainte. Une exploration à l'avant et une à l'arrière du graphe permet de calculer le nombre total de chemins (donc des solutions de la contrainte) et la densité des solutions.

Les algorithmes de dénombrement proposés forment la base d'une nouvelle famille d'heuristiques décrites dans la section suivante.

Heuristiques basées sur le dénombrement

Malgré les nombreux efforts de recherche pour concevoir des heuristiques de recherche génériques et robustes et pour analyser leurs comportements, les applications qui utilisent la PPC nécessitent souvent des heuristiques conçues autour du problème, ou au moins des ajustements de celles qui sont standard et génériques. De l'autre coté, les solveurs de Programmation Linéaire en Nombre Entiers ou de SAT offrent avec succès des heuristiques

de recherche par défaut qui, fondamentalement, réduisent le problème à un effort de seule modélisation. La PPC par contre n'offre pas encore des heuristiques qui soient génériques, robustes et aussi efficaces. Cette contribution vise à faire des étapes importantes dans cette direction.

On propose d'extraire des informations de dénombrement de solutions des contraintes et de guider de la recherche en exploitant cette information. En particulier, on vise à explorer des parties de l'arbre de recherche qui probablement contiennent un pourcentage élevé de solutions. La plus simple mais aussi plus efficace heuristique proposée (MAXSD) choisit la paire variable-valeur avec la plus grande densité de solution. On a introduit de nombreuses variations des heuristiques basées sur le dénombrement, cherchant d'utiliser l'information provenant de plusieurs contraintes à travers différentes fonctions d'agrégation.

Les résultats expérimentaux (plus de 6000 heures de temps de calcul) portent sur huit domaines différents (allant de problèmes d'ordonnancement sportif aux problèmes de confection d'horaire pour le personnel et aux certains problèmes communs de référence dans la communauté scientifique). On a comparé nos heuristiques avec certaines heuristiques considérées comme étant l'état-de-l'art. L'analyse a montré des résultats très prometteurs pour les heuristiques proposées. En particulier, l'heuristique MAXSD a été en mesure de résoudre en moyenne près de 97 % des exemplaires. A titre de comparaison, la meilleure heuristique à laquelle on s'est comparé n'a réussi à résoudre que 82% des cas, en prenant en moyenne presque 4 fois plus de temps de calcul.

Pour conclure, les heuristiques basées sur le dénombrement améliorent de manière significative la résolution par rapport aux autres heuristiques génériques, permettant donc de résoudre plus efficacement les problèmes ou même pour résoudre des problèmes plus gros. La quantité de calcul ajoutée pour dénombrer les solutions de contraintes est donc bien compensée par le gain en terme d'efforts pour la recherche d'une solution. Ce qu'on propose peut être une étape importante vers une procédure de recherche entièrement automatisée, générique et aussi efficace.

Amélioration du Quick Shaving

Des formes fortes de cohérence se sont révélées être un élément efficace pour bien adresser les problèmes combinatoires difficiles avec la programmation par contraintes. L'attention a été récemment concentrée sur une notion de cohérence plus puissante appelée cohérence Singleton (Singleton Consistency). La cohérence singleton garantit qu'une affectation d'une variable à une valeur ne conduit pas à un échec immédiat après que le réseau de contraintes a

été propagé. Malgré que la cohérence singleton sans aucun doute réduit sensiblement l'espace de recherche, il n'est toujours pas accepté univoquement si cette réduction de l'espace de recherche récompense l'effort mis pour l'obtenir. Ce qui rend la cohérence singleton particulièrement lourde, c'est l'affectation temporaire d'une variable à une valeur et à la propagation du réseau de contraintes qui s'ensuit seulement pour vérifier si la valeur est cohérente. De plus cet effort est souvent fait pour plusieurs (voir toutes) paires variable-valeur et potentiellement même plusieurs fois. Le temps de calcul supplémentaire n'est récompensé par une certaine réduction de l'espace de recherche que lorsque l'affectation temporaire conduit à un échec.

Afin de réduire davantage l'effort computationnel dû à la cohérence singleton, d'autres approches avec une puissance d'inférence réduite ont été développées. En particulier, des chercheurs ont proposé le Quick Shaving, une forme réduite de cohérence singleton qui, au lieu de travailler de façon proactive à chaque noeud de l'arbre de recherche, ne vérifie que de façon réactive les paires variable-valeur qui ont récemment causé un échec.

Cette méthode, malgré qu'elle soit sensiblement plus légère que la cohérence singleton, n'assure quand même pas un gain en terme de temps de résolution.

On a étudié le comportement du quick shaving pour comprendre si son taux de réussite (combien de fois une tentative de shaving se traduit concrètement par un filtrage du domaine) est corrélée à une amélioration en performance. On a remarqué une forte corrélation entre les deux. Chaque fois que le taux de réussite est plus élevé qu'un seuil donné un gain de performance est attendu. Étonnamment, le seuil est bien défini et cohérent pour tous les exemplaires testés (neuf domaines de problème et six heuristiques).

Par la suite, on a introduit une fonctionnalité qui mesure le taux de réussite et s'il n'atteint pas le minimum nécessaire on désactive dynamiquement le quick shaving. Cet algorithme a permis au quick shaving d'être essayé et subitement désactivé au cas où il n'amène pas des améliorations de performance, sans devoir payer l'effort additionnel du quick shaving tout au long de l'arbre de recherche; cependant là où il y a un avantage de calcul important il reste activé pour aider dans la réduction de l'espace de recherche. L'algorithme proposé peut élargir l'utilisation des techniques basées sur le quick shaving, en enlevant de l'utilisateur final la charge de sélectionner et tester manuellement le niveau de cohérence de solveurs.

Contents

Dedication . . . iii

Abstract . . . iv

Résumé . . . v

Condensé en Français . . . vi

Contents . . . xiii

List of Tables . . . xvi

List of Figures . . . xvii

List of Annexes . . . xix

CHAPTER 1 Introduction . . . 1
1.1 Background . . . 3
1.1.1 Elements of Graph Theory . . . 3
1.1.2 Elements of Constraint Programming . . . 6

CHAPTER 2 Filtering Algorithms . . . 13
2.1 Soft Global Cardinality Constraint . . . 13
2.1.1 Soft Global Cardinality Constraint . . . 14
2.1.2 Soft `gcc` and Matching . . . 16
2.1.3 Consistency and Filtering Algorithms . . . 19
2.2 Generalizations of the Global Cardinality Constraint for Hierarchical Resources 23
2.2.1 Nested Global Cardinality Constraint . . . 24
2.2.2 Further generalization . . . 29
2.2.3 Experimental results . . . 32
2.2.4 Softening the Nested Global Cardinality Constraint . . . 34
2.2.5 Expressing preferences . . . 36
2.3 Summary . . . 38

CHAPTER 3 Counting Algorithms 39
3.1 Counting for alldifferent Constraints 40
3.1.1 Computing the Permanent 41
3.1.2 Rasmussen's Estimator and Its Extensions 43
3.1.3 Upper Bounds 46
3.1.4 Counting Accuracy Analysis 50
3.1.5 Extending permanent upper bounds to the symmetric alldifferent constraint 53
3.2 Counting for Global Cardinality Constraint 54
3.3 Counting for Regular Constraints 58
3.3.1 Counting Paths in the Associated Graph 59
3.3.2 An Incremental Version 61
3.3.3 A Lazy Evaluation Version 61
3.4 Summary 62

CHAPTER 4 Solution Counting Based Heuristics 63
4.1 Background 64
4.2 Generic Constraint-Centered Counting-based Heuristics 69
4.3 Experimental Analysis 77
4.3.1 Experimental Settings 77
4.3.2 Quasigroup completion problem with holes 79
4.3.3 Nonograms 86
4.3.4 Multi dimensional knapsack problem 89
4.3.5 Market Split Problem 91
4.3.6 Magic Square Completion Problem 93
4.3.7 Cost-Constrained Rostering Problem 95
4.3.8 Rostering Problem 97
4.3.9 Travelling Tournament Problem with Predefined Venues 99
4.4 Summary and conclusions 105

CHAPTER 5 Experimental Analysis of Quick Shaving 110
5.1 Quick Shaving 111
5.2 Understanding Quick Shaving Behaviour 114
5.2.1 Problems 115
5.2.2 Heuristics 115
5.2.3 Experimental Analysis 116
5.3 Dynamically Disabling Quick Shaving 121

5.4 Conclusion 123

CHAPTER 6 Conclusions 124

Bibliography 126

Annexes 136

List of Tables

Table 2.1	Experimental results for `nested_gcc`; time are expressed in seconds .	34
Table 4.1	Average results for 40 hard QWH instances	80
Table 4.2	Average results for 40 hard QWH instances with randomized restarts	82
Table 4.3	Average solving time (in seconds) and number of backtracks for 100 QWH instances of order 30.	84
Table 4.4	Average results for 180 Nonogram instances	87
Table 4.5	Average results for 25 Multi Knapsack instances	90
Table 4.6	Average results for 10 Market Split instances	92
Table 4.7	Average results for 40 Magic Square instances	94
Table 4.8	Average results for 10 Cost-Constrained Rostering instances	96
Table 4.9	Average results for 60 Rostering instances	98
Table 4.10	Average results for 40 TTPPV balanced instances (model 1)	101
Table 4.11	Average results for 40 TTPPV non-balanced instances (model 1) . . .	102
Table 4.12	Average results for 40 TTPPV balanced instances (model 2)	104
Table 4.13	Average results for 40 TTPPV non-balanced instances (model 2) . . .	105
Table 4.14	Average solving time (in seconds) and number of backtracks for 40+40 balanced and non balanced instances of the TTPPV.	107
Table 4.15	Aggregated average results over the eight problem domains	108
Table 5.1	Arithmetic average solving time (in seconds), number of backtracks and percentage of solved instances without quick shaving	117
Table 5.2	ϕ correlation coefficient	121
Table .1	Average results for 40 hard Latin Square instances	137
Table .2	Average results for 180 Nonogram instances	138
Table .3	Average results for 25 Multi Knapsack instances	139
Table .4	Average results for 10 Market Split instances	140
Table .5	Average results for 40 Magic Square instances	141
Table .6	Average results for 10 Cost Rostering instances	142
Table .7	Average results for 60 Rostering instances	143
Table .8	Average results for 40 TTPPV balanced instances (model 1)	144
Table .9	Average results for 40 TTPPV non-balanced instances (model 1) . . .	145
Table .10	Average results for 40 TTPPV balanced instances (model 2)	146
Table .11	Average results for 40 TTPPV non-balanced instances (model 2) . . .	147
Table .12	Aggregated average results over the eight problem domains	148

List of Figures

Figure 1.1 Solution Process 7
Figure 1.2 Value ordering impact. 8
Figure 2.1 (a) GCC bipartite graph (for each value, upper and lower bound are indicated between parenthesis). (b) Maximum Matching in G_o. (c) Maximum Matching in G_u. (d) Possible solution with minimum violation. 17
Figure 2.2 Traditional GCC modelling for the Nested_GCC 26
Figure 2.3 (a) `nested_gcc` Graph Representation for Example 5: if not shown the lower and upper bounds are respectively null and unitary. (b) Schematic graph representation for Example 5. 27
Figure 2.4 Skill level relations: (a) linearly ordered skill levels and (b) tree-like ordered skill levels. 30
Figure 2.5 (a) Programmers skill relations. (b) Requirements for component c_1. (c) Requirements for component c_2. 30
Figure 2.6 (a) Resource relation. (b) Constraint graph representation. 32
Figure 2.7 (a) `soft_nested_gcc` Graph Representation for Example 5: dashed arcs have unitary cost and bold arcs are used by one possible min-cost maximum flow. (b) Schematic graph representation for Example 5. . 36
Figure 2.8 Graph representation for the nested gcc with preferences. 38
Figure 3.1 Counting Error for one thousand `alldifferent` instances with varying variable domain sizes. 51
Figure 3.2 Maximum Solution Density Error for one thousand `alldifferent` instances with varying variable domain sizes. 52
Figure 3.3 Average Solution Density Error for one thousand `alldifferent` instances with varying variable domain sizes. 52
Figure 3.4 Lower Bound Graph (a) and Residual Upper Bound Graph (b) for Example 8 56
Figure 3.5 Lower Bound Graph (a) and Residual Upper Bound Graph (b) assuming $x_1 = 1$ 58
Figure 3.6 The layered directed graph built for a `regular` constraint on five variables. Vertex labels represent the number of incoming and outgoing paths. 59
Figure 4.1 Percentage of solved instances vs time (in seconds) for QWH 80

Figure 4.2 Percentage of solved instances vs time (in seconds) for QWH with randomized restarts . . . 82
Figure 4.3 Percentage of solved instances vs time (in seconds) for QWH instances with 42% of holes . . . 85
Figure 4.4 Percentage of solved instances vs time (in seconds) for QWH instances with 45% of holes . . . 85
Figure 4.5 Total average time vs % holes . . . 86
Figure 4.6 Percentage of solved instances vs time (in seconds) for the Nonogram problem . . . 88
Figure 4.7 Percentage of instances solved vs time (in seconds) for the Multi Knapsack problem . . . 91
Figure 4.8 Percentage of instances solved vs time (in seconds) for the Market Split problem . . . 93
Figure 4.9 Percentage of instances solved vs time (in seconds) for the Magic Square problem . . . 94
Figure 4.10 Percentage of instances solved vs time (in seconds) for Cost-Constrained Rostering problem . . . 97
Figure 4.11 Percentage of instances solved vs time (in seconds) for Rostering problem 99
Figure 4.12 Percentage of non-balanced instances solved vs time for TTPPV problem (first model - TTPPV1) . . . 103
Figure 4.13 Percentage of non-balanced instances solved vs time for TTPPV problem (second model - TTPPV2) . . . 106
Figure 4.14 Comparison of maxSD heuristic with different counting algorithms: percentage of solved instances vs time in seconds for non balanced instances of the TTPPV. . . . 107
Figure 5.1 Quick Shaving Example . . . 114
Figure 5.2 Time Ratio: average time with QS over average time without QS . . . 118
Figure 5.3 Average Shaving Ratio . . . 119
Figure 5.4 Scattered plots . . . 120
Figure 5.5 Scattered plots . . . 121
Figure 5.6 Time Ratio: average time of QS with shaving filter over average time without QS . . . 122

LIST OF ANNEXES

Full Experimental Results 136

CHAPTER 1

Introduction

Constraint satisfaction problems (CSPs) are mathematical problems where one has to find states or objects in a system that satisfy a number of constraints or criteria. Constraints have emerged as a research area that combines researchers from a number of fields, including Artificial Intelligence, Programming Languages, Symbolic Computing and Computational Logic. Constraint networks and constraint satisfaction problems have been studied since the seventies.

The field of Constraint Programming (CP) arose from Programming Languages and Logic Programming in the 1980's and nowadays is an efficient paradigm to solve combinatorial and optimization problems. Constraint Programming has been successfully applied to numerous domains; recent applications include computer graphics, natural language processing, database systems, molecular biology, business applications, transport applications, logistics, supply-chain and inventory problems, workforce management optimization, resource scheduling, electrical engineering, circuit design and so on.

Global constraints are a key aspect of CP because they encapsulate problem substructures: they ease the modelling process for the user and more importantly they have allowed a dramatic improvement of the solution process by avoiding unnecessary work during the exploration of possible problem solutions. Offering new global constraints or improving the constraint reasoning algorithms of already known constraints have therefore a direct impact on the capabilities and efficiency that CP offers to the user.

However, constraint reasoning is often not enough to solve real-life problems, therefore search is needed to evaluate different paths that may lead to the problem solutions. Different generic search heuristics have been developed over the years, nonetheless the user still faces the choice between spending resources and time to develop a highly efficient problem-specific heuristic or relying on generic search heuristics with the risk of poor performance. On this front other paradigms (Mixed Integer Linear Programming for example) offer to the user efficient heuristics embedded in the solver that can be used directly out-of-the-box; in this respect, CP lags behind and it still does not provide a completely automated generic yet efficient search heuristic. Each step towards this direction is crucial to broaden the use of CP.

This thesis revolves around these two topics: constraint reasoning (filtering algorithms) for global constraints and search heuristics. On one side we improve existing filtering algorithms

and propose a new global constraint; on the other, we introduce a completely novel way to design search heuristics that exploits in a more integrated fashion the global constraints of the model by counting the number of solutions the individual constraints admit. This new family of heuristics aims to be generic and yet efficient to solve real-life problems. Finally, we touch a topic that lies in between the two previous ones, that is a strong form of constraint reasoning that is heuristically guided by the search (quick shaving).

More specifically, the main contributions are:

- a new improved filtering algorithm for one of the most common constraints, that is `soft_gcc` (appeared in [111])
- a new global constraint that is a generalization of the `gcc` and that embeds the concept of heterogeneous resources for assignment problems (appeared in [112])
- counting algorithms for the `alldifferent`, `symmetric_alldifferent`, `gcc` and `regular` constraints (partially appeared in [113], [114] and [116])
- counting-based heuristics that extract information on the number of solutions of the model constraints and steer the search toward the parts of the search tree that are likely to contain a high number of solutions (partially appeared in [113], [114])
- an improved quick shaving technique, that is a strong form of constraint reasoning guided by search (appeared in [115])

The contributions proposed have a direct impact on solving real-life problems. Particularly, the better time complexity and better efficacy of the filtering algorithms proposed will allow to tackle larger and more complex problem instances that are now out-of-reach. Furthermore, the ease of implementation of the `soft_gcc` algorithm turned out to be a winning point as it is now implemented in a commercial solver [1] whereas the previously known algorithm is not present in any commercial solver at the time of writing.

Counting-based heuristics go in the direction of improving both efficiency and ease of use of solvers for the end-users. On one side, they significantly improve over other generic heuristics allowing therefore to solve more efficiently problems or even to address larger problem instances. On the other side, it is a step toward a completely automated, generic yet efficient search as it can be found in other SAT or MIP solvers.

Finally, the improved quick shaving will possibly broaden the use of shaving techniques by lifting the end-user the burden of fine tuning the consistency level of solvers. The proposed technique in fact automatically trigger this strong form of constraint reasoning only when it is needed and it can actually bring a benefit to solve the problem in hand.

The thesis is organized as follows: in the next section, we will briefly give a common background on Graph Theory and CP that will be used throughout the following chapters;

1. Comet^{TM}by Dynadec - www.dynadec.com

Chapter 2 presents new filtering algorithms for two global constraints; in Chapter 3, we will introduce counting algorithms for the global constraints mentioned above; in Chapter 4, we will present a new family of efficient heuristics based on solution counting; in Chapter 5, we will experimentally examine quick shaving and we will improve it by introducing a simple yet effective adaptive approach to better exploit such technique. Finally, conclusions will be drawn in Chapter 6.

1.1 Background

1.1.1 Elements of Graph Theory

In this section we recall the main results and definitions that will be used in the next sections (see [1] for further explanations).

A graph is defined as $G = (V, E)$ where V is a set of vertices and E is a set of unordered pairs (edges) from V. A graph is called bipartite if V can be partitioned in two subset X and Y and all the edges are in the form $e = \{v_i, v_k\}$ where $v_i \in X$ and $v_j \in Y$ (i.e. there is no edge that joins two vertices of the same subset). A graph $G' = (V', E')$ is defined as a *subgraph* of $G = (V, E)$ if $V' \subseteq V$, $E' \subseteq E$ and V' contains all the vertices adjacent to the edges in E'. A path in a graph $G = (V, E)$ is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$, $i = 0, \ldots, k-1$. A graph is called *connected* iff for each vertex pair $u, v \in V$ there exists a path between u and v. A *connected component* or *component* of a graph $G = (V, E)$ is a connected subgraph G' of G such that no other connected subgraph contains G'.

An oriented graph is defined as $G = (V, A)$ where V is a set of vertices and A is a set of ordered pairs (arcs) from V. We write $\delta^{out}(v)$ to refer to the set of outgoing arcs of v: $\delta^{out}(v) = \{(v, u) \mid (v, u) \in A\}$. Similarly, the set of ingoing arcs of v is denoted by $\delta^{in}(v) = \{(u, v) \mid (u, v) \in A\}$. An oriented path in an oriented graph $G = (V, A)$ is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in A$, $i = 0, \ldots, k-1$. An oriented graph is called a *strongly connected component* iff for each ordered pair (u, v) of vertices there exists an oriented path from u and v. A *strongly connected component* of an oriented graph $G = (V, A)$ is a strongly connected subgraph G' of G such that no other strongly connected subgraph contains G'.

Matching Theory

Definition 1 (Maximum Matching)**.** *A subset of edges in a graph G is called* matching *if no two edges have a vertex in common. A matching of maximum cardinality is called a maximum matching.*

Given a matching M in G, a vertex is called a *free vertex* if it is not adjacent to any edge of the matching M.

An alternating path with respect to a matching M (*M-alternating path*) is defined as a path whose edges $e_i = (v_i, v_{i+1})$ belong alternatively to $E - M$ and to M.
An augmenting path with respect to a matching M (*M-augmenting path*) is defined as a path that starts from and ends at a free vertex, and its edges $e_i = (v_i, v_{i+1})$ belong alternatively to $E - M$ (odd edges) and to M (even edges); note that an augmenting path has an odd number of edges.
Intuitively, an augmenting path can be used to increase the number of edges that belong to a matching. Given a matching M and an M-augmenting path P, we can build M' as $M' = M \oplus P$ (the set operation $\oplus$ is defined as $A \oplus B = (A - B) \cup (B - A)$), that is the odd numbered edges are added to the matching and the even numbered edges are removed from the matching; the resulting matching increases its cardinality, $|M'| = |M| + 1$.

Theorem 1 (Petersen [80])**.** *A matching M is maximum if and only if there is no augmenting path relative to M.*

Theorem 2. *Let G be a graph and M a maximum matching in G. An edge belongs to a maximum matching in G if and only if it either belongs to M, or to an even M-alternating path starting from a free vertex, or to an even alternating circuit.*

Lemma 1. *Given a maximum matching M in G, for any edge $e = (v_i, v_j)$ in G, there exists a matching M_e such that $e \in M_e$ and $|M_e| \geq |M| - 1$*

Proof. If e belongs to M then $M_e = M$; otherwise, starting from the matching M, we obtain M_e adding e and removing all the edges that belong to M and that are incident to v_i or v_j (at most one on each). The result is a matching of size $|M_e| \geq |M| + 1 - 2$. □

We introduce the concept of degree $deg_M(v)$ of a vertex v as the number of edges adjacent to v that belongs to M (for the traditional definition of matching $deg_M(v) \in \{0, 1\}$).

Theorem 3. *Given a matching M in G, an M-augmenting path P and the matching $M' = M \oplus P$, each vertex v has $deg_{M'}(v) \geq deg_M(v)$.*

Proof. The degree of a vertex v decreases if and only if v is not free w.r.t M and the incident edge that belongs to M is removed from the matching. For every removed edge $e = (v, v_j)$, two new edges from P are added, incident respectively to v and v_j, so $deg_M(v) = deg_{M'}(v)$. □

Hopcroft and Karp (see [53]) described an algorithm based on Theorem 1 with a running time complexity of $O(\sqrt{n}m)$ where n is the number of vertices and m the sum of the cardinalities of the domains.

In [83], Quimper et al. generalized this algorithm maintaining the same complexity. In their generalization they associate to each vertex of the graph a capacity. Given a matching M, the capacity of a vertex v indicates the maximum number of edges in M adjacent to v.

Intuitively they build a duplicated graph G_d in which every vertex with a capacity greater than one is substituted by a number of vertices equal to the capacity, also the edges associated to these vertices are duplicated. In this way a traditional matching (in which all the capacities are equal to 1) in G_d corresponds to a matching on the original graph (in which the capacities can be greater than 1).

Quimper's approach is equivalent to the traditional one when all the capacities are set to 1.

Network Flows

Let $G = (V, A)$ be an oriented graph, $l(a)$ and $c(a)$ the demand and the capacity of each arc $a \in A$ $(0 \leq l(a) \leq c(a))$. We define *s-t flow* as a function $f : A \to \mathbb{R}$ such that:

$$\forall v \in V \setminus \{s, t\} : \sum_{\{a \in \delta^{out}(v)\}} f(a) = \sum_{\{a' \in \delta^{in}(v)\}} f(a')$$

where s and t are respectively the *source* and *sink* of the flow. The flow is feasible if $\forall a \in A : l(a) \leq f(a) \leq c(a)$. The value of a flow f is defined as $value(f) = \sum_{\{a \in \delta^{out}(s)\}} f(a) - \sum_{\{a' \in \delta^{in}(s)\}} f(a')$. A feasible flow f is maximum if there is no other feasible flow f' such that $value(f') > value(f)$.

Theorem 4. *If all arc demands and capacities are integer and there exists a feasible flow, then the maximum flow problem has an integer maximum flow.*

Let $w : A \to \mathbb{R}$ be a cost function. The cost of a flow f is defined as:

$$cost(f) = \sum_{a \in A} w(a) f(a)$$

The minimum cost flow is the flow with minimum cost among the feasible flows. The *successive shortest path algorithm* [1] can find a minimum cost flow with a time complexity $O(n \cdot L \cdot SP)$ where n is the number of vertices of the graph, L the largest arc demand and SP is the time to compute the shortest path on G.

1.1.2 Elements of Constraint Programming

A *Constraint Satisfaction Problem* (CSP) consists of a finite set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ with finite domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ such that $x_i \in D_i$ for all i, together with a finite set of constraints $\mathcal{C}$, each on a subset of $\mathcal{X}$. A constraint $C \in \mathcal{C}$ is a subset $T(C)$ of the Cartesian product of the domains of the variables that are in C. We write $X(C)$ to denote the set of variables involved in C and we call tuple $\tau \in T(C)$ an allowed combination of values of $X(C)$. The number of occurrences of a value d in a tuple τ is denoted by $\#(d, \tau)$. An assignment $(d_1, \ldots, d_k) \in X(C)$ satisfies a constraint C if it belongs to $T(C)$. A feasible *solution* to a CSP is an assignment of all the variables such that it satisfies all the constraints.

A *Constraint Optimization Problem* (COP) is a CSP with an associated objective function to be minimized or maximized. A solution to the CSP is also a solution (possibly sub-optimal) to the COP. An optimal solution of a minimization (resp. maximization) COP has an objective value that is not higher (resp. lower) to any other feasible solution of the COP.

Constraint Programming (CP) is an efficient paradigm to solve CSPs and COPs. CP is commonly seen as the composition of *modelling* and *search*. Declarative modelling defines the set of variables involved in the CSP and the associated constraints; search defines how to explore the space of (possibly infeasible) solutions. Constraint inference (also referred to as propagation or filtering) allows to reduce the search space as it removes parts of the search space that are proven to be infeasible.

We will describe in the following, from a high-level perspective, what is considered to be the most efficient CP algorithm to find a solution to a CSP, that is Maintaining Arc Consistency (MAC) (see [93] and [12]).

The solution process proceeds by iteratively interleaving search phases and propagation phases. During the search phase, generally performed on a tree-like structure, a variable is instantiated to a value of its domain. Then, in the propagation phase, each constraint checks its consistency (i.e. whether it is feasible or not). In case the constraint admits no solution, it fails and backtrack occurs; otherwise, constraint inference is performed and reflected on variable domains. Constraint inference removes values from the variable domains that are inconsistent w.r.t. the partial assignment built so far. Every time a constraint reduces a variable domain, the other constraints that are registered on that variable have to propagate again until the fixed point is reached, that is, no further filtering can be inferred (see [26]). If, while achieving the fixed point, one of the variable domains becomes empty then the search fails and it backtracks to reconsider the branching decision. After achieving the fixed point, a new search step is performed. The solution process finishes when a solution is found, that is, a value is assigned to each variable, or when one of the following conditions is achieved:

the tree has been fully explored without finding a solution, a time or a backtrack limit has been reached.

Example 1. *Given four variables x_1, x_2, x_3, x_4 defined on the domains $D_1 = D_2 = \{1, \ldots, 5\}$, $D_3 = \{2, 3, 4\}$ and $D_4 = \{2, \ldots, 7\}$. The constraint set is $C_1 : x_1 \neq x_2$, $C_2 : x_1 \geq x_3$, $C_3 : x_1 < x_4$ and $C_4 : x_3 \neq x_4$. The search process is shown in Figure 1.1.*

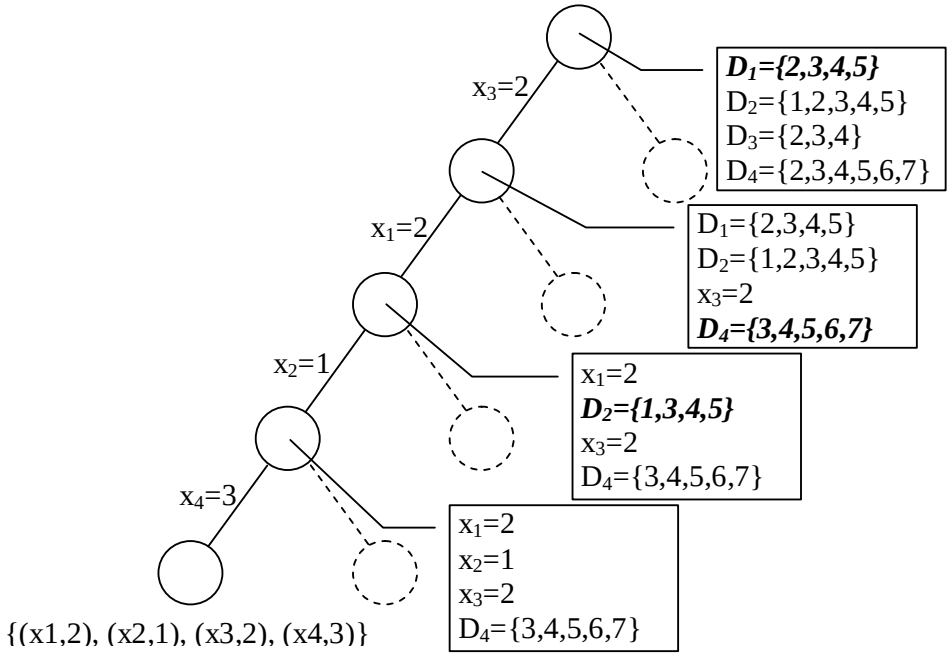


Figure 1.1 Solution Process

- *Initial propagation - C_2 propagates and removes the value 1 from D_1 since it does not have any support in D_3.*
- *Instantiation - $x_3 = 2$.*
- *Propagation - C_4 propagates and removes the value 2 from D_4.*
- *Instantiation - $x_1 = 2$.*
- *Propagation - C_1 propagates and removes the value 2 from D_2.*
- *Instantiation - $x_2 = 1$.*
- *Propagation - No propagation required.*
- *Instantiation - $x_4 = 3$.*
- *Solution found - $\{(x_1 = 2), (x_2 = 1), (x_3 = 2), (x_4 = 3)\}$*

Both in search and propagation phases, some fundamental aspects (that will be discussed in the following sections) deeply affect the performance of the solution process.

Search Phase

During the search phase, the choice of the variable to instantiate next is crucial as well as the value to assign to the variable. These two choices are referred to as *variable selection heuristics* and *value selection heuristics*.
We give a brief example to emphasize the impact of the variable selection heuristic on the search process.

Example 2. *Suppose we have an infeasible problem defined on the variable set* $\{x_1, \ldots, x_n\}$. *Suppose moreover that there is an inconsistency (not detected by the propagation) between the variables* x_{n-1} *and* x_n. *If the variable heuristic chooses sequentially from* x_1 *up to* x_n *then the tree will have exponential size and it will look like Figure 1.2(a). On the other hand, if variables* x_{n-1} *and* x_n *are the first to be instantiated then the tree will look like Figure 1.2(b) giving a much smaller tree that can be explored more quickly .*

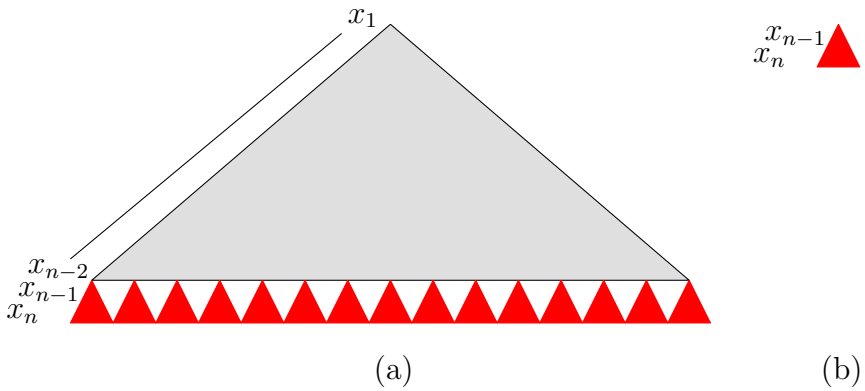


Figure 1.2 Value ordering impact.

Besides variable selection heuristics, value selection heuristic comes into play. In infeasible problems, the value selection heuristic does not affect the performance since the whole tree must be explored anyway. Nonetheless, if we need a single solution or an optimal solution among a set of feasible solutions, then it becomes relevant. The value heuristic can focus more on feasibility or on optimality; in the former, it tries to keep a high probability to reach a solution (for example we branch on the value that keeps the highest possible Cartesian product of the variable domains); in the latter, we branch depending on costs or reduced costs of the values. Generally, for the value ordering a custom problem-dependent heuristic is used (see Chapter 4 for an in-depth literature review on search heuristics).

Once variable and value heuristics are chosen, the shape of the tree is defined. It still remains to decide which search procedure will be used i.e. how the tree will be explored. We briefly mention the most known search procedures: Depth-First Search (DFS), Iterative

Deepening (ID) [63], Limited Discrepancy Search (LDS) [52], Depth-Bounded Discrepancy Search (DDS) [108], Discrepancy-Bounded Depth First Search (DBDFS) [7].

Propagation Phase

The propagation phase is fundamental to reduce the size of the search space and to try to avoid exploring an exponential size space. During the propagation, different degrees of consistency can be achieved and they differ mainly on the impact they have on the reduction of the domains.

Definition 2 (Arc Consistency)**.** *Given a binary constraint C defined on two variables x_1 and x_2 with respective domains D_1 and D_2, the constraint is* arc consistent *iff for each value $d_1 \in D_1$ there exists a value $d_2 \in D_2$ such that $(d_1, d_2) \in T(C)$ and for each value $d_2 \in D_2$ there exists a value $d_1 \in D_1$ such that $(d_1, d_2) \in T(C)$.*

Definition 3 (Bounds Consistency)**.** *Given a constraint C defined on the variable set $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, the constraint is* bounds consistent *iff for each variable x_i and each value $d_i \in \{min\ D_i, max\ D_i\}$ there exists a value $d_j \in [min\ D_j, max\ D_j]$ for all $j \neq i$ such that $(d_1, \ldots, d_n) \in T(C)$.*

Definition 4 (Range Consistency)**.** *Given a constraint C defined on the variable set $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, the constraint is* range consistent *iff for each variable x_i and each value $d_i \in D_i$ there exists a value $d_j \in [min\ D_j, max\ D_j]$ for all $j \neq i$ such that $(d_1, \ldots, d_n) \in T(C)$.*

Definition 5 (Domain Consistency)**.** *Given a constraint C defined on the variable set $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, the constraint is* domain consistent *iff for each variable x_i and each value $d_i \in D_i$ there exists a value $d_j \in D_j$ for all $j \neq i$ such that $(d_1, \ldots, d_n) \in T(C)$.*

Note that domain consistency is also referred to as *hyper-arc consistency* or *generalized arc consistency* (GAC). It can be proved that a constraint that achieves Domain Consistency reduces the domains at least as much as one that achieves Range Consistency; then we say that Range Consistency is weaker than Domain Consistency. Bound Consistency is weaker than Range Consistency.

Global Constraints

We now introduce a classification of the constraints involved in a CSP: *non-decomposable constraints* and *global constraints*. Non-decomposable constraints are constraints that cannot

be expressed by a set of other simpler constraints while global constraints [2] are a conjunction of a set of other constraints (either non-decomposable or global). Even if every CSP with global constraints can be translated to an equivalent one with binary non-decomposable constraints, the introduction of global constraints has proved to be fundamental to speed-up the solution process. Global constraints allow ease of expressiveness and special purpose filtering algorithms can be implemented to possibly increase the amount of propagation. Generally, specific filtering algorithm achieve domain consistency while consistency on the equivalent set of non-decomposable constraints cannot guarantee domain consistency.

Example 3. *Let x_1, x_2, x_3 be three variables defined on the domain $D_1 = D_2 = \{1, 2\}$ and $D_3 = \{1, 2, 3\}$. The constraint set is: $\{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\}$ and it can be expressed through the global constraint* `alldifferent`*. Enforcing local consistency on the constraint set does not allow to remove any value from the domain: each single constraint is arc consistent. However, thanks to a global reasoning, we can infer that variables x_1 and x_2 cover the values 1 and 2 so they can be removed from D_3.*

Propagation algorithms: two case studies In this section we will review the propagation algorithms of the `alldifferent` and `gcc` as they will be part of the focus of our work.

The `alldifferent` constraint has been successfully applied to a wide variety of problems and it is considered one of the most relevant global constraints.

Definition 6. *Given a set of variables $X = \{x_1, \ldots, x_n\}$ with respective domains $D_1, \ldots, D_n$, we define the* `alldifferent` *as:*

$$T(C) = \{(d_1, d_2, \ldots, d_n) \mid d_i \in D_i, d_i \neq d_j \ \ for\ i \neq j\}$$

Régin proposed a propagation algorithm based on matching theory in the seminal paper [88].

Definition 7 (Value Graph)**.** *Given a set of variables $X = \{x_1, \ldots, x_n\}$ with respective domains $D_1, \ldots, D_n$, we define the* value graph *as a bipartite graph $G = (X \cup D_X, E)$ where $D_X = \bigcup_{i=1,\ldots,n} D_i$ and $E = \{\{x_i, d_i\} \mid d_i \in D_i\}$.*

There exists a bijection between a maximum matching with $|M| = |X|$ on the value graph and a solution of the related `alldifferent` constraint. An edge $e = (x_i, d_i)$ in a matching M represents an assignment $x_i = d_i$. Since the edges of a matching by definition do not

2. An alternative definition is: *"a global constraint is a constraint that captures a relation between a non-fixed number of variables"* (from [92])

have any vertex in common hence a maximum matching with $|M| = |X|$ represents a feasible assignment for the `alldifferent` constraint.

Theorem 5 (Domain Consistency)**.** *An* `alldifferent` *constraint defined on the variable set X is domain consistent iff all the edges in the related value graph belong to at least one maximum matching M with $|X| = |M|$.*

The propagation algorithm proceeds in two phases: it firstly checks if the constraint is feasible i.e. there exists at least a solution that satisfies it. Afterwards, it achieves domain consistency removing all the edges that do not belong to any maximum matching. For the consistency check, it searches a maximum matching M using Hopcroft-Karp algorithm [53]. If one exists with $|X| = |M|$ then the constraint admits at least one feasible solution otherwise it fails. For the filtering, it identifies all the edges belonging to at least a maximum matching exploiting Theorem 2. Technically, the alternating paths starting from a free vertex are found through a breadth-first search; note that during this search each edge is traversed at most once. To find the alternating circuits, it builds an oriented graph $G' = (V, A)$ with $A = \{(x_i, d_i) \,|\, \{x_i, d_i\} \in M\} \cup \{(d_i, x_i) \,|\, \{x_i, d_i\} \notin M\}$; then it searches the strongly connected components on G' that correspond to alternating circuits on G. All the edges traversed neither by an alternating path nor by an alternating circuit can be removed (and consequently values from their respective variable domains) since they do not belong to any maximum matching. The filtering algorithm has a complexity of $O(\sqrt{n}m)$ for finding the maximum matching and of $O(m)$ and $O(n + m)$ for finding respectively alternating paths and alternating circuits, where $n = |V|$ and $m = |E| = \sum_{i=1}^{n} |D_i|$. Thus, the overall complexity is $O(\sqrt{n}m)$.

The Global Cardinality Constraint – $\texttt{gcc}(X, l, u)$ is a generalization of the `alldifferent` proposed by Régin in [89]. Instead of allowing the assignment of each value at most once, it constrains more generally the number of occurrences of each value in a solution.

Definition 8. *Let $X = \{x_1, \ldots, x_n\}$ be a set of variables with respective domains $D_1, \ldots, D_n$, and l and u two vectors containing the minimum and maximum number of occurrences for each value $d \in D_X$. We define the* `gcc` *as:*

$$T(C) = \{\tau \mid \tau \in X(C),\, \forall d \in D_X : l_d \leq \#(d, \tau) \leq u_d\}$$

where $\#(d, \tau)$ is the number of occurrences of the value d in the tuple τ.

Initially, Régin proposed a filtering algorithm based on a network flow algorithm with a complexity of $O(nm)$ [89]. Later on, C-G. Quimper et al. in [83] improved this result

presenting an algorithm inspired by that of the `alldifferent` with a complexity equal to $O(\sqrt{n}m)$. Their main result was that achieving domain consistency on the upper bound constraint ($\forall d \in D_X : \#(d, \tau) \leq u_d$) and on the lower bound constraint ($\forall d \in D_X : l_d \leq \#(d, \tau)$) imply the domain consistency on the `gcc`.
They define the lower bound value graph in which variable vertices have unitary capacities and value vertices have capacities equal to their respective lower bounds. Analogously, for the upper bound value graph, value vertices have capacities equal to their respective upper bounds.

Theorem 6 (Domain Consistency)**.** *A* `gcc` *constraint defined on the variable set X is domain consistent iff all the edges belong at least to a maximum matching M with $|X| = |M|$ in the related lower and upper bound value graphs.*

The algorithm proceeds in a similar way the `alldifferent` propagating algorithm does. It finds a maximum matching on the capacitated value graphs and then it checks which edges belong either to an alternating path or to an alternating circuit. The ones that do not belong to maximum matching either in the lower bound graph or in the upper bound graph are then removed.

CHAPTER 2

Filtering Algorithms

In this chapter, we present in Section 2.1 a pair of new algorithms for propagating the `soft-global-cardinality-constraint` which improve the complexity over previously known algorithms. In Section 2.2 we introduce a new constraint that is an extension of the `global-cardinality-constraint`. Finally, we wrap up with a brief summary in Section 2.3. Portions of this chapter appeared in [111] and [112].

2.1 Soft Global Cardinality Constraint

Many real-life problems are over-constrained. The tightness and the high number of constraints can make the problems become infeasible. In these situations it is worth finding a solution that partially violates some constraints but that it is still interesting for the user. Constraints can be partitioned among *hard constraints* that cannot be violated, and *soft constraints* that can be (partially) violated. Hard constraints are used for modelling the inherent structure of the problem and soft constraints are more related to preferences that the user wishes to introduce to the model. Clearly, solutions satisfying a maximum of preferences are more interesting for the user. Different approaches deal with the concept of violation in different ways: some methods (MAX-CSP) try to minimize the number of violated constraints, others (Weighted-CSP [64] [65], Possibilistic-CSP [95], Fuzzy-CSP [27] [29]) propose more fine-grained ways to measure the level of violation. Petit et al. in [81] proposed a new approach in which the over-constrained problem is translated to a traditional constraint optimization problem; this allows to reuse all the code base and expertise developed for COP. Petit proposed to introduce, for each soft constraint, a cost variable representing the violation and an associated function that measures the violation of the constraint. The main objective is then to find a solution that minimizes the total violation that is a function of the individual constraints' violations; common examples are the sum of all the violations or the maximum. As for traditional constraint optimization problems, it is then worth trying to identify special purpose filtering algorithms that can prune the variable domains on the basis of the cost (violation). Recent work started in this direction by exploring the area of soft global constraints. In particular van Hoeve et al. in [107] exploited Flow Theory and proposed filtering algorithms for the soft versions of the well known `alldifferent`, `gcc`, `regular`, and `same` constraints.

In this section we present an improved algorithm for achieving generalized arc consistency for the soft `gcc` (with variable based violation and value based violation) exploiting Matching Theory, with a better complexity. Intuitively the soft `gcc` constraint is violated when either

- too many variables are assigned to a value, exceeding its upper bound (producing an overflow) or
- too few variables are assigned to a value, violating its lower bound (producing an underflow) or
- both.

The idea is to compute separately the best possible overflow and underflow and, since we claim they are independent, find a class of solutions minimizing both overflow and underflow. On the basis of these best underflow and overflow we perform filtering.

This work on the soft `gcc` is organized as follows: in Section 2.1.1 we formally present the soft `gcc` constraint and the related violation measures; then we discuss the relationship between the violation measures and matching theory. In Section 2.1.2 we introduce the consistency theorems and the filtering algorithms for reaching generalized arc consistency.

2.1.1 Soft Global Cardinality Constraint

A *Global Cardinality Constraint* on a set of variables specifies the minimum and the maximum number of occurrences for each value in a solution.

Definition 9 (Global Cardinality Constraint)**.**

$$T(gcc(X,l,u)) = \{(d_1,d_2,\ldots,d_n) \mid d_i \in D_i, l_d \leq |\{d_i | d_i = d\}| \leq u_d \, \forall d \in D_X\}$$

A generic definition for a soft version of the gcc is:

Definition 10 (Soft Global Cardinality Constraint)**.**

$$T(softgcc[*](X,l,u,z)) = \\ \{(d_1,d_2,\ldots,d_n,d_z) \mid d_i \in D_i,\ d_z \in D_Z, violation_{softgcc[*]}(d_1,d_2,\ldots,d_n) \leq d_z\}$$

*where * defines a violation measure for the gcc.*

To calculate the violation measures van Hoeve et al. (see [107]) introduced the following definitions:

Definition 11. *Given a $softgcc(X,l,u,z)$, we define for all $d \in D$*

$$overflow(X,d) = max(|\{x_i \mid x_i = d\}| - u_d, 0)$$

$$underflow(X,d) = max(l_d - |\{x_i \mid x_i = d\}|, 0)$$

Definition 12 (Variable-based violation)**.** *Given a constraint C and a solution $\tilde{X}$, the* variable-based violation *is defined as the number of variables that should change their value in order to satisfy C.*

Lemma 2 (SoftGCC Variable-based violation)**.** *Given a softgcc, if* $\sum_{d\in D_X} l_d \leq |X| \leq \sum_{d\in D_X} u_d$ *then the* variable based violation *can be expressed as:*

$$violation_{[var]}(X) = max\Big(\sum_{d\in D} overflow(X,d), \sum_{d\in D} underflow(X,d)\Big)$$

Consider for example the variables x_1, x_2, x_3, and x_4 and the related domains $D_1 = \{1,2\}$, $D_2 = \{1,2\}$, $D_3 = \{1,2\}$, and $D_4 = \{1,2,3\}$. Suppose we post the constraint $\texttt{softgcc}[\texttt{var}](\{x_1,x_2,x_3,x_4\},\{l_1 = 0, l_2 = 1, l_3 = 2\}, \{u_1 = 1, u_2 = 1, u_3 = 2\}, z)$. A possible assignment is $(1,1,2,3)$ which has an overflow equal to 1 and an underflow equal to 1; the variable-based violation is equal to 1.

Note that it is not possible to calculate the variable based violation whenever $\sum_{d\in D_X} l_d \leq |X| \leq \sum_{d\in D_X} u_d$. To avoid this limitation van Hoeve et al. introduced the *value-based violation* (see [107]):

Definition 13 (Value-based violation)**.** *Given a softgcc, the* value-based violation *is defined as:*

$$violation_{[val]}(X) = \sum_{d\in D} overflow(X,d) + \sum_{d\in D} underflow(X,d)$$

Consider again the example mentioned above. The assignment $(1,1,2,3)$ which has unitary overflow and underflow, has a value-based violation equal to 2.

Van Hoeve et al. (see [107]) proposed two algorithms (one for variable-based violation and one for value-based violation) achieving generalized arc consistency both based on flow theory. In their solution they build a value graph (similarly to Régin in [90]) in which some arcs take into account the violations; a cost is associated to each of these arcs. A maximum flow with minimum cost in that graph is equivalent to a solution with minimum violation of the soft gcc.
Their algorithms have a complexity of $O(n(m + n\log n))$ for variable-based violation and of $O((n+k)(m+n\log n))$ for value-based violation (k is the cardinality of the union of the domains).

2.1.2 Soft gcc and Matching

The main idea of this section is to exploit matching theory to calculate two assignments that minimize respectively the overflow and the underflow. We prove that it is possible to find a class of assignments that have overflow and underflow equal to the respective bounds. Then, we figure out how the violation cost of this class of assignments may change when we force an individual assignment $x_i = d$. Finally, we can perform filtering based on optimality reasoning.

Let $G(X \cup D, E)$ be an undirected bipartite graph (also called value graph) such that one partition represents the variable set and the other one the value set. There is an edge $(x_i, d) \in E$ if and only if $d \in D_i$.

Overflow Let G_o be a value graph such that the capacities of value-vertices are set to $c(d) = u_d$ (variable-vertices have unitary capacity) and we assume $u_d \geq 1$. Using the algorithm described in Section 1.1.1, we compute a maximum matching M_o in G_o. A maximum matching M_o corresponds to an assignment that should satisfy the upper bound constraint of the gcc. If $|M_o| = |X|$ then the matching corresponds to a consistent assignment (w.r.t. the upper bound constraint); if $|M_o| < |X|$ it means that some variables cannot be assigned to a value otherwise the upper bound constraint would be violated.
Exactly $|X| - |M_o|$ variables must be assigned to some values that have already reached the maximum number of occurrences so the overflow is exactly $|X| - |M_o|$.

Theorem 7. *Given a maximum matching M_o in the graph G_o, it is not possible to find an assignment with a total overflow less than $|X| - |M_o|$.*

Proof. Suppose that there exists an assignment X with an overflow equal to $OF < |X| - |M_o|$. We build the bipartite graph that represents X and we remove from this graph the OF edges that cause the overflow, therefore each value-vertex d has $deg(d) \leq u_d$. The resulting graph can be seen as a feasible matching M' in G_o. Since $|M'| = |X| - OF$ then $|M'| > |M_o|$, i.e. M_o is not maximum. □

In Figure 2.1 we give an example of the concept explained above. Figure 2.1a shows the value graph of a global cardinality constraint; the variable domains are $D_1 = D_2 = D_3 = D_4 = D_5 = \{v_1, v_2\}$, $D_6 = \{v_3, v_4\}$, $D_7 = \{v_2, v_3\}$, $D_8 = D_9 = \{v_4, v_5\}$; for each value the minimum and the maximum number of occurences are indicated between parenthesis. Figure 2.1b shows G_o and the related maximum matching. In details, the maximum matching M_o in G_o has an overflow equal to $|X| - |M_o| = 1$; as we can see x_5 causes the overflow since it is not assigned w.r.t. M_o.

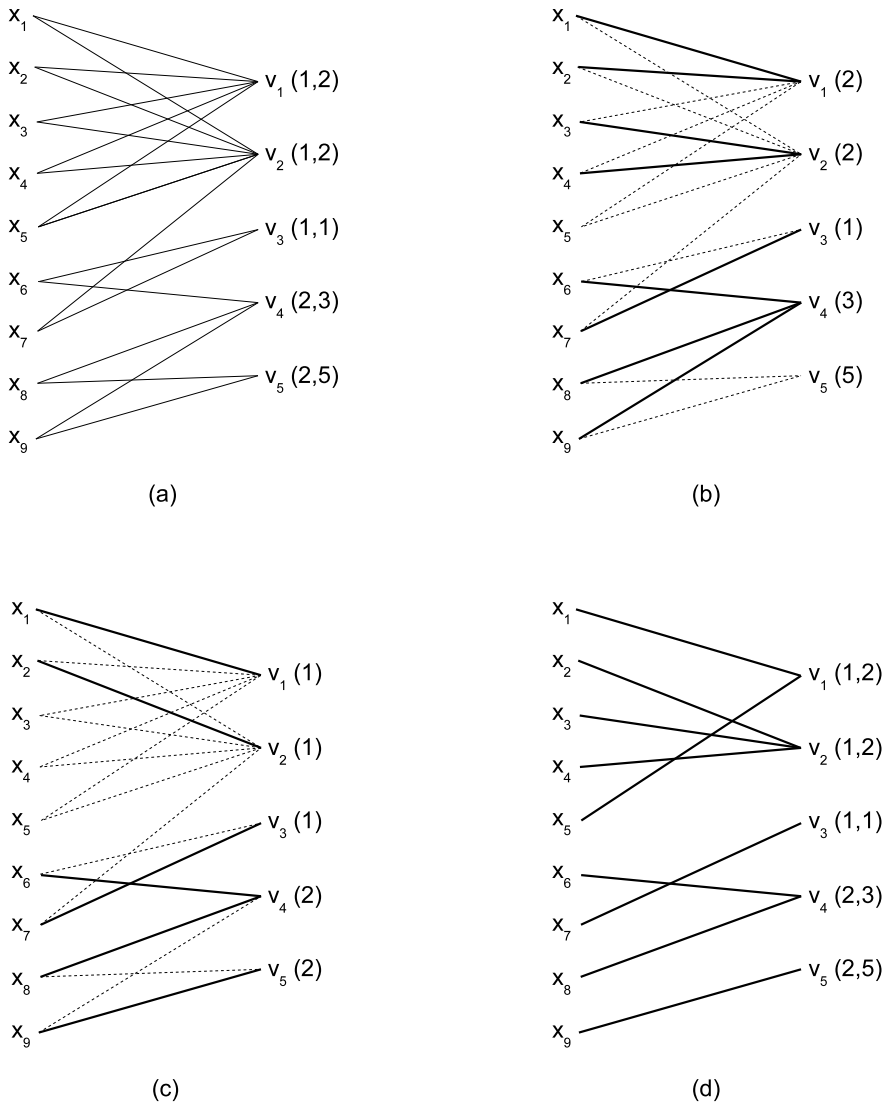


Figure 2.1 (a) GCC bipartite graph (for each value, upper and lower bound are indicated between parenthesis). (b) Maximum Matching in G_o. (c) Maximum Matching in G_u. (d) Possible solution with minimum violation.

Underflow Analogously, we exploit matching theory to compute the underflow. In this case the graph G_u is built such that the capacities of value-vertices are set to $c(d) = l_d$ (variable-vertices have unitary capacity), assuming $l_d > 0$ [1]. A maximum matching M_u in G_u corresponds to a partial assignment that should satisfy the lower bound constraint of the gcc.

If $|M_u| = \sum_{d \in D} l_d$ then it means that for each value $deg_{M_u}(d) = l_d$, thus there exists at least

1. This assumption has been relaxed in: P. Schaus, P.Van Hentenryck and A. Zanarini "Revisiting the Soft Global Cardinality Constraint", to appear in CPAIOR-2010

one (partial) assignment that satisfies the lower bound constraint (i.e. there is no underflow and no violation w.r.t. the lower bound constraint). If $|M_u| < \sum_{d \in D} l_d$ then there are one or more values that do not reach the minimum number of requested occurrences (some value vertices are still free) and there are no variables that are unmatched and that can be assigned to these values.
Note that $l_d - deg_{M_u}(d) \geq 0$, hence by definition:

$$underflow(X, d) = l_d - deg_{M_u}(d)$$

and the total underflow is:

$$\sum_{d \in D} underflow(X, d) = \sum_{d \in D} l_d \ - deg_{M_u}(d) =$$

$$= \sum_{d \in D} l_d \ - \sum_{d \in D} deg_{M_u}(d) = \sum_{d \in D} l_d - |M_u|$$

Figure 2.1c shows G_u and the related maximum matching. The maximum matching M_u in G_u has an underflow equal to $\sum_{d \in D} l_d - |M_u| = 1$ that is caused by the value v_5.

Theorem 8. *Given a softgcc constraint and two maximum matchings M_o and M_u, respectively in G_o and G_u, it is possible to build a class of assignments with overflow equal to $BOF = |X| - |M_o|$ (best overflow) and underflow equal to $BUF = \sum_{d \in D} l_d - |M_u|$ (best underflow).*

Proof. We compute a maximum matching M_u in G_u whose underflow is equal to BUF. The matching M_u is clearly a feasible matching (probably not maximum) also in G_o because all the capacities of G_o are greater than those of G_u. Starting from M_u we compute the maximum matching M_o in G_o whose overflow is equal to BOF. As stated in Theorem 3, when we compute a matching, the degree of each vertex does not decrease, hence the underflow of M_o in G_o remains equal to BUF.
If $|M_o| < |X|$ then there exists a set X_{OF} of unassigned variables, that is, there is no edge in M_o adjacent to the variables in X_{OF}. These variables cause the overflow and, in the final solution, can be assigned to any value in their respective domain. □

Figure 2.1d shows how it is possible to find a solution that minimizes overflow and underflow: this assignment has a variable-based violation equal to 1 and a value-based violation equal to 2.

In order to develop a filtering algorithm, it is worth figuring out how overflow and underflow may change (w.r.t. the bounds of Theorem 8) when we try to force an individual assignment $x_i = d$. They change depending on whether the edge (x_i, d) belongs to a maximum matching in the graphs G_o and G_u or not; intuitively if it belongs to a maximum matching the overflow (or underflow) does not change otherwise it increases by 1.

Theorem 9. *Given a softgcc constraint, an individual assignment $x_i = d$ and a solution $\tilde{X}$ with $x_i = d$ that minimizes the overflow (OF) and the underflow (UF) then $BOF \leq OF \leq BOF + 1$ and $BUF \leq UF \leq BUF + 1$ where BOF is the best overflow and BUF the best underflow.*

Proof. Let G_o and G_u be the overflow and underflow graphs and M_o and M_u the related maximum matchings. Suppose we remove from G_o (overflow graph) and G_u (underflow graph) the vertex x_i (and the related edges) and decrease u_d and l_d by 1; we call the resulting graph G'_o and G'_u. This is equivalent to forcing $x_i = d$ in the final assignment. Then we find the maximum matching M'_o in G'_o and M'_u in G'_u, clearly their cardinalities can be at most $|M'_o| = |M_o| - 1$ and $|M'_u| = |M_u| - 1$. Hence:

- if $|M'_o| = |M_o| - 1$ and $|M'_u| = |M_u| - 1$ then $x_i = d$ belongs to a maximum matching both in G_o and in G_u and the assignment has $OF = BOF$ and $UF = BUF$;
- if $|M'_o| = |M_o| - 1$ and $|M'_u| < |M_u| - 1$ then $x_i = d$ belongs to a maximum matching in G_o but not in G_u and the assignment has $OF = BOF$ and $UF = BUF + 1$ (equivally if $x_i = d$ belongs to a maximum matching in G_u but not in G_o);
- if $|M'_o| < |M_o| - 1$ and $|M'_u| < |M_u| - 1$ then $x_i = d$ does not belong to a maximum matching in G_o nor in G_u and the assignment has $OF = BOF + 1$ and $UF = BUF + 1$.

□

2.1.3 Consistency and Filtering Algorithms

In this section we explain the basis to reach generalized arc consistency and we show the filtering algorithms for the variable-based and value-based violations. Our approach is similar to the one proposed by Petit et al. in [81] for the `soft_alldifferent` constraint. We assume for the `soft_gcc` the lower bounds to be strictly positive.

Briefly, we recall that the variable z represents the cost of the violation and D_z its domain; during the search $max\ D_z$ represents the maximum violation allowed; the objective is to minimize z in order to minimize the total violation.
Moreover, we recall that variable-based violation is equal to

$max\Big(\sum_{d\in D} overflow(X,d), \sum_{d\in D} underflow(X,d)\Big)$ and that value-based violation is equal to $\sum_{d\in D} overflow(X,d) + \sum_{d\in D} underflow(X,d)$.

Variable Based Violation

Theorem 10. *Let G_o and G_u be the value graphs with respectively upper and lower bound capacities and let M_o and M_u be maximum matchings respectively in G_o and G_u; let BOF and BUF be respectively $BOF = |X| - |M_o|$ and $BUF = \sum_{d\in D} l_d - |M_u|$. The constraint $softgcc_{[var]}(X,l,u,z)$ is generalized arc consistent on X if and only if $\min D_z \leq max(BOF\ ,\ BUF)$ and either:*

1. $\max\Big(\mathrm{BOF}\ ,\ \mathrm{BUF}\Big) \leq (\max \mathrm{D_z} - 1)$ *or*
2. *if* $\Big(\mathrm{BOF} = \max \mathrm{D_z}\Big)$ *and* $\Big(\mathrm{BUF} \leq (\max \mathrm{D_z} - 1)\Big)$ *and all edges in G_o belong to a maximum matching or*
3. *if* $\Big(\mathrm{BOF} \leq (\max \mathrm{D_z} - 1)\Big)$ *and* $\Big(\mathrm{BUF} = \max \mathrm{D_z}\Big)$ *and all edges in G_u belong to a maximum matching or*
4. *if* $\Big(\mathrm{BOF} = \mathrm{BUF} = \max \mathrm{D_z}\Big)$ *and all edges in G_u and in G_o belong to a maximum matching.*

Proof. In the first case we can build an assignment with $violation_{[var]}$ strictly less than $max\ D_z$; from Theorem 9 the change of a single variable can cause a unitary increase of the overflow and underflow hence the total violation is still less or equal to $max\ D_z$.
In the second case (resp. third case) if the overflow (resp. underflow) is equal to $max\ D_z$ then all the edges must belong to a maximum matching in G_o (resp. in G_u) such that there is no violation increase; from Theorem 9 we know that an edge that does not belong to a maximum matching would cause an overflow (resp. underflow) increase making it greater than $max\ D_z$.
In the last case the only way to not have a violation increase is that all edges belong to a maximum matching both in G_o and in G_u. □

Filtering algorithm Firstly we compute the maximum matchings M_o in G_o and M_u in G_u. If the overflow or underflow is greater than $max\, D_z$ then we fail because the best possible solution is worse than the maximum allowed violation.
If $BOF = |X| - |M_o| < max\ D_z$ and $BUF = \sum_{d\in D} l_d - |M_u| < max\ D_z$ then all the values are consistent.
In the case of $|X| - |M_o| = max\ D_z$ we can remove all the edges that do not belong to a

maximum matching in G_o; from matching theory (Theorem 2), we know that an edge can be part of a matching iff it belongs to a strongly connected component (alternating circuit) or it lies on an alternating path of even length starting from or leading to a free vertex. Analogously, if $\sum_{d \in D} l_d - |M_u| = max\ D_z$ we remove all the edges that do not belong to a maximum matching in G_u. Finally, we update the bound of the violation variable, if necessary ($min\ D_z = max(BOF\ ,\ BUF)$).

The maximum matchings can be computed in $O(\sqrt{n}m)$ through Quimper's adaptation of Hopcroft-Karp's algorithm (where n is the number of variables and m the sum of the cardinalities of the domains); the running time for computing strongly connected components is $O(n+m)$ and for finding alternating paths is $O(m)$, hence the overall complexity can be bounded by $O(\sqrt{n}m)$.

Note that if all the values have lower bounds equal to 0 then the filtering on the lower bound graph is not necessary has no underflow can occur. Analogously if the values have upper bounds greater or equal to the number of variables then the filtering on the upper bound graph can be avoided (as no overflow can occur).

In the speacial case where all the values have u_d equal to 1, the `soft_gcc` is equivalent to the `soft_alldifferent` and the algorithm proposed correspond to Petit et al.'s solution (see [81]).

Consider a `soft_gcc` with four variables and three values and suppose that $max\ D_z = 1$. Variable domains are $D_1 = D_2 = D_3 = \{1, 2\}$, $D_4 = \{1, 2, 3\}$ and the values $(1, 2, 3)$ have lower bounds and upper bounds respectively of $(1, 1, 2)$ and $(1, 1, 4)$. Firstly we compute a maximum matching in G_o: $M_o = \{(x_1, 1), (x_2, 2), (x_4, 3)\}$; thus the overflow is $OF = |X| - |M_o| = 1$. Then we compute a maximum matching in G_u: $M_u = \{(x_1, 1), (x_2, 2), (x_4, 3)\}$; the underflow is $UF = \sum_{d \in D} l_d - |M_u| = 1$. Since both the overflow and the underflow are equal to $max\ D_z$ then we prune all the edges that do not belong to a maximum matching in G_o and/or in G_u. In particular, all the edges belong to a maximum matching in G_o; the edges $(x_4, 1)$ and $(x_4, 2)$ do not belong to a maximum matching in G_u, so they can be pruned; in fact if x_4 would have been equal to 1 (or 2) then the underflow would have been equal to 2 (caused by the value 3).

Value Based Violation

Theorem 11. *Let G_o and G_u be the value graphs with respectively upper and lower bound capacities and let M_o and M_u be maximum matchings respectively in G_o and G_u; let BOF and BUF be respectively $BOF = |X| - |M_o|$ and $BUF = \sum_{d \in D} l_d - |M_u|$. The constraint $softgcc_{[val]}(X, l, u, z)$ is generalized arc consistent on X if and only if $min D_Z \leq BOF + BUF$*

and either:

1. $\mathrm{BOF} + \mathrm{BUF} < (\max \mathrm{D_Z} - 1)$ *or*
2. *if* $\mathrm{BOF} + \mathrm{BUF} = (\max \mathrm{D_Z} - 1)$ *and either all edges belong to a maximum matching at least in one of* G_o *or* G_u *or*
3. *if* $\mathrm{BOF} + \mathrm{BUF} = \max \mathrm{D_Z}$ *and all edges belong to a maximum matching both in* G_o *and in* G_u.

Proof. We start from the best solution found following Theorem 8. From this solution a single change of a variable can cause in the worst case a violation increase equal to 2 (Theorem 9). So in the worst case the total violation is less than or equal to $max\, D_Z$; hence all the values are consistent.
If the overall violation is equal to $max\, D_Z - 1$ then we have to verify that all the edges belong to at least a maximum matching; for Theorem 9 the maximum violation increase would be at most equal to 1, hence the total violation remains less or equal to $max\, D_Z$.
If the overall violation is equal to $max\, D_Z$ and all the edges belong to a maximum matching in both G_o and G_u then there would be no increase in the total violation so the constraint remains feasible. □

Filtering algorithm Firstly we compute the maximum matchings M_o in G_o and M_u in G_u.
If $(OF + UF)$ is greater than $max\, D_z$ then we fail because the best possible solution is worse than the best current solution found.
If $(OF + UF) < max\, D_z - 1$ then all the values are consistent. In the case of $(OF + UF) = max\, D_z - 1$ we can remove all the values that do not satisfy at least one of the following conditions: belonging to a maximum matching in G_o or belonging to a maximum matching in G_u.
If $(OF + UF) = max\, D_z$ then we remove all the edges that do not belong to a maximum matching in G_o and/or in G_u.
Finally, we update $min\, D_z$, if necessary ($min\, D_z = (OF + UF)$).

The overall complexity is analogous to the variable-based algorithm, that is $O(\sqrt{n}m)$.

Following the example shown in Figure 2.1, suppose that $max\, D_z = 3$.
Instead, if we consider the value-based violation, we have to remove all the edges that belong neither to a maximum matching in G_o nor in G_u. In particular focusing on G_o, the edges $e_1 = (x_7, v_2)$ and $e_2 = (x_6, v_3)$ belong neither to an alternating circuit nor to an alternating path starting from or leading to a free vertex. This means that they do not belong to a

maximum matching in G_o. Analyzing G_u, the situation is analogous. Hence, e_1 and e_2 cause an increase equal to 2 of the total violation (unitary increase of overflow and of underflow). Forcing e_1 (or e_2) to be in a solution, the resulting value-based violation is 4 then e_1 (resp. e_2) is inconsistent and can be pruned.

2.2 Generalizations of the Global Cardinality Constraint for Hierarchical Resources

Resource allocation problems occur in many real-life problems whenever it is necessary to assign resources to tasks that need to be accomplished. It can be thought of as a one to one assignment or, more generally, a many to one relation in which tasks can be assigned one or more resources. Typically, for each task a minimum and maximum number of required resources is defined. Resources may be *homogeneous* in the sense that they have identical capabilities or skills. In constraint programming, problems with homogeneous resources can be easily modeled by a global cardinality constraint [89] (`gcc`) in which each resource is represented by a variable whose domain is the set of tasks and each task defines its resource requirements through the bounds on the number of occurrences in the definition of the constraint. However for some real-world problems this scenario is too simplistic: resources are heterogeneous and tasks require resources with different capabilities or skill levels. We further distinguish three cases: in the first, referred to as *disjoint heterogeneous resources*, the different skill levels are considered independently i.e. a resource with a given skill level can only satisfy requirements defined on this level; in the second, referred to as *nested heterogeneous resources*, resources are organized in a total order, that is, a resource with skill level ℓ is able to satisfy requirements of level ℓ or below; in the third, which we call *hierarchical heterogeneous resources*, the relationship between resources generalizes beyond the linear order of the nested case to a tree-like hierarchy.

Problems with disjoint heterogeneous resources are also easily modeled, this time using a set of `gcc`'s, each of them representing a single skill level as before, and domain consistency can still be achieved. Unfortunately a similar model for the nested and hierarchical cases does not guarantee domain consistency. This section focuses on the important cases of nested and hierarchical heterogeneous resources for which we propose generalizations of the global cardinality constraint that achieve domain consistency.

Example 4. *We need to accomplish concurrently two tasks $T1$ and $T2$ that have different requirements of resources of level 1 and 2 (level 2 is higher than level 1, meaning that resources of level 2 are more skilled). Three resources R_1^1, R_2^1, R_3^1 of level 1 and three resources R_1^2, R_2^2, R_3^2 of level 2 are available. Each resource can be assigned to any task. Both tasks $T1$ and $T2$*

need between 1 and 2 resources of level 2, and between 2 and 3 resources of level 1. Assigning the appropriate number of resources to the tasks, while satisfying the level requirements, allow to correctly accomplish the task.

In a disjoint heterogeneous resources setting, resources can only satisfy requirements of their level. Since the tasks need at least 4 resources of level 1 the problem is unsatisfiable.

In a nested heterogeneous resources setting, resources can satisfy requirements of their level or below. The minimum requirements of resources of level 2 is equal to 2 (one for each task). Then, one resource of level 2 can be assigned to a task for satisfying the requirements of level 1. Thus, the problem is satisfiable.
Note that in the case of nested heterogeneous resources the problem can be restated as follows: both tasks $T1$ and $T2$ need respectively between 3 and 5 resources of level 1 or higher, and among them 1 or 2 must be of level 2.

The initial motivation for this work came from a real-life manpower planning and scheduling problem proposed in [91] by France Telecom for the 2007 ROADEF Challenge. A subproblem consists of forming teams of technicians that have to accomplish a set of tasks. The technicians have different skill levels and a technician can satisfy task requirements of his level or below. Each task defines the minimum number of technicians required for each skill level. This corresponds exactly to nested heterogeneous resources. Another important application area is nurse rostering. Here a minimum number of nurses on duty is specified for each work shift and sometimes a minimum is also given for senior nurses acting in a supervisory role but who can perform the duties of regular nurses as well. Applications of hierarchical heterogeneous resources are found in the computer software industry or generally in large projects with multiskilled resources.

This section is organized as follows: in Section 2.2.1, we formally introduce the `nested_gcc`, its graph representation as well as the theoretical basis for achieving domain consistency. Section 2.2.2 is dedicated to a generalization of the `nested_gcc` called `hierarchical_gcc`. In Section 2.2.3 we show experimental evidence of the usefulness of the presented constraints. Section 2.2.5 considers an optimization version of `nested_gcc` that allows the expression of preferences.

2.2.1 Nested Global Cardinality Constraint

In the following, we write $\tau \downarrow_{X'}$ for the projection of the tuple τ over the set X'. The number of occurrences of a value d in a tuple τ is denoted by $\#(d, \tau)$; analogously $\#(d, \tau \downarrow_{X'})$ is the number of occurrences of d in the projection of the tuple τ over the set X'.

Let $\{X^k\}_{1 \leq k \leq \ell}$ represent a family of ℓ disjoint sets of variables. Define further $\mathbb{X}^k = \bigcup_{k \leq j \leq \ell} X^j$, with $\mathbb{X} = \mathbb{X}^1$ for short. Observe that this new family of sets is nested: $\mathbb{X}^\ell \subseteq$

$\mathbb{X}^{\ell-1} \subseteq \cdots \subseteq \mathbb{X}^1$. The variables $X^k = \{x_1^k, \ldots, x_{n_k}^k\}$ are defined over the domains $D_1^k, \ldots, D_{n_k}^k$. We write D_{X^k} for the union of the domains of the variables in X^k; analogously $D_{\mathbb{X}}$ stands for the union of all the domains of the variables in $\mathbb{X}$.
We denote by l_d^k and u_d^k the lower and upper bounds on the number of occurrences of value $d \in D_{\mathbb{X}}$ among $\mathbb{X}^k$. It follows that we should have $l_d^{k+1} \leq l_d^k$ and $u_d^{k+1} \leq u_d^k$ for $k = 1, \ldots, \ell-1$. For example $l_d^1 = 5$, $u_d^1 = 7$, $l_d^2 = 3$, $u_d^2 = 4$ means that value d occurs between 5 and 7 times in $\mathbb{X}^1$, including between 3 and 4 times in $\mathbb{X}^2$. The related vectors of lower and upper bounds are denoted by l^k and u^k for $k = 1, \ldots, \ell$.

Definition 14 (nested gcc)**.** *The nested global cardinality constraint is formally defined as*

$$T(\texttt{nested\_gcc}(\texttt{X}^1, \ldots, \texttt{X}^\ell, (\texttt{l}^1, \texttt{u}^1), \ldots, (\texttt{l}^\ell, \texttt{u}^\ell))) =$$

$$\{\tau = (d_1^1, \ldots, d_{n_1}^1, d_1^2, \ldots, d_{n_\ell}^\ell) \mid d_i^k \in D_i^k, \forall 1 \leq k \leq \ell, \forall d \in D_{\mathbb{X}} : l_d^k \leq \#(d, \tau \downarrow_{\mathbb{X}^k}) \leq u_d^k\}$$

Note that it is possible to model the `nested_gcc` as a set of traditional `gcc`'s: for each set $\mathbb{X}^k$, we define a `gcc` in which we set the corresponding upper and lower bounds. But such a formulation is strictly weaker, as we shall see in Proposition 1.

Going back to our resource allocation problem, the tasks would correspond to the values and the resources to the variables, arranged in disjoint sets according to their level. As defined, the constraint requires that each resource be assigned to a task. In some problems, like rostering, it might be useful to find an assignment that, while satisfying the lower bound constraints, keeps some resources unassigned. This can be easily modeled by adding an extra value (without requirements) representing a fake activity; resources that are assigned to it are in fact unused.

Example 5. *A company needs to form some teams in order to accomplish 5 tasks. Only 6 technicians with different skills are available. Three of them have skill level equal to 2 (capable of accomplishing a job requiring skill level 1 or 2) and the remaining three have a skill level equal to 1. Moreover the three technicians with basic skills are not allowed to be assigned to the task 5. We model the problem with 6 variables representing the resources divided in two sets:* $X^1 = \{x_1^1, x_2^1, x_3^1\}$ *and* $X^2 = \{x_1^2, x_2^2, x_3^2\}$*. The variable domains are respectively:* $D_1^1 = D_2^1 = D_3^1 = \{d_1, d_2, d_3, d_4\}$ *and* $D_1^2 = D_2^2 = D_3^2 = \{d_1, d_2, d_3, d_4, d_5\}$*. The tasks require respectively a minimum of 1, 1, 1, 1 and 2 technicians of skill level at least 1. Tasks 3 and 4 each need at least one technician of level 2. None of the tasks can accommodate more than 3 technicians (independently from the level). We would model this situation as* $\texttt{nested\_gcc}(\{\texttt{x}_1^1, \texttt{x}_2^1, \texttt{x}_3^1\}, \{\texttt{x}_1^2, \texttt{x}_2^2, \texttt{x}_3^2\}, ([1,1,1,1,2], [3,3,3,3,3]), ([0,0,1,1,0], [3,3,3,3,3]))$*. The alternate model using two* `gcc`*'s is illustrated in Figure 2.2.*

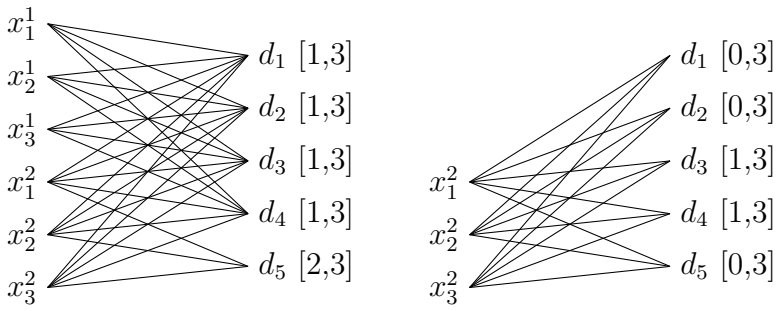


Figure 2.2 Traditional GCC modelling for the Nested_GCC

Proposition 1. *Modelling the constraint* `nested_gcc` *as a set of traditional* `gcc` *does not achieve domain consistency.*

Proof. Consider Example 5. Both `gcc` constraints are domain consistent however the instance is unsatisfiable. Two variables in X^2 must take the value d_3 and d_4 (from the level 2 `gcc`), hence there is only one variable left to assign to d_5 that requires a minimum of two variables (from the level 1 `gcc`). □

Even though it has been proven ([28]) that finding a consistent solution to a set of overlapping `gcc`'s is an NP-Complete problem, the particular nested structure is such that it is possible to find a consistent assignment in polynomial time as we shall see in the next section.

Graph Representation

We propose a new graph representation for the `nested_gcc`. Informally, it contains vertices representing the variables from the X^k sets and vertices that denote the values; differently from the traditional `gcc`, the value vertices are duplicated for each set X^k while variable vertices remain singletons; arcs connect successive replications of value vertices. In order to identify duplicate value vertices, for each value $d_i \in D_{\mathbb{X}}$ we add a superscript that refers to its corresponding set X^k. We write $D_{\mathbb{X}}^k$ to denote the set of values in $D_{\mathbb{X}}$ with superscript k; hence $d_i^k \in D_{\mathbb{X}}^k$ and $d_i^{k'} \in D_{\mathbb{X}}^{k'}$ represent the value d_i but two different value vertices for sets X^k and $X^{k'}$. The directed graph $G = (V, A)$ is defined as follows:

$$V = \mathbb{X} \cup \big(\bigcup_{k=1}^{\ell} D_{\mathbb{X}}^k\big) \cup \{s, t\}$$

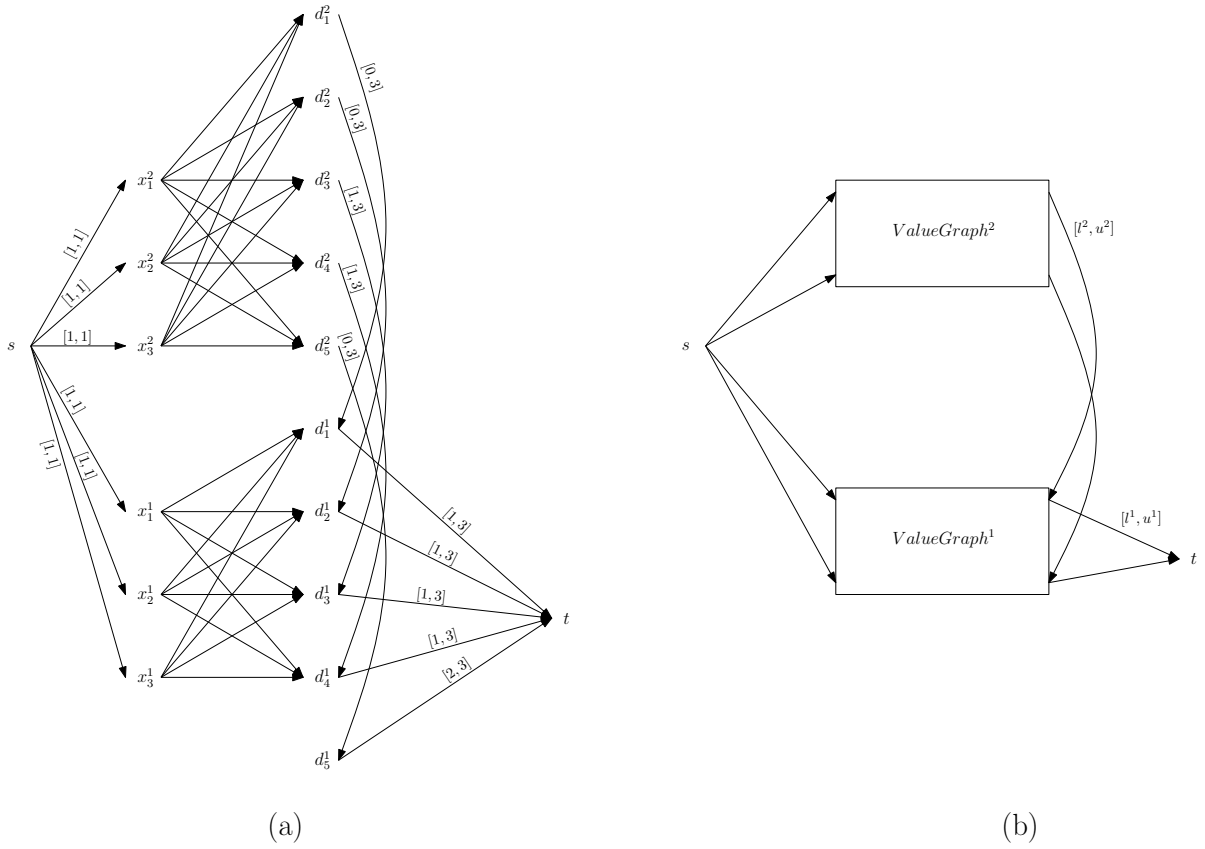
$$A = A_s \cup \Big(\bigcup_{k=1}^{\ell} A_{X^k}\Big) \cup \Big(\bigcup_{k=1}^{\ell} A^k_{req}\Big)$$

where

$$\begin{aligned} A_s &= \{(s, x_i^k) \mid k = 1, \ldots, \ell\,,\ i = 1, \ldots, n_k\} \\ A_{X^k} &= \{(x_i^k, d_j^k) \mid i = 1, \ldots, n_k, d_j^k \in D_i^k\} \\ A^k_{req} &= \begin{cases} \{(d_i^1, t) \mid i = 1, \ldots, |D_{\mathbb{X}}|\} & \text{if } k = 1 \\ \{(d_i^k, d_i^{k-1}) \mid i = 1, \ldots, |D_{\mathbb{X}}|\} & \text{if } 2 \le k \le \ell \end{cases} \end{aligned}$$

The lower bounds and upper bounds of the arcs $a \in A_s$ are unitary; they are respectively null and unitary for the arcs $a \in A_{X^k}$. For each arc (d_i^k, v) the lower bound is equal to l_i^k and the upper bound is u_i^k.

The graphical representation of Example 5 is given in Figure 2.3.



(a) (b)

Figure 2.3 (a) `nested_gcc` Graph Representation for Example 5: if not shown the lower and upper bounds are respectively null and unitary. (b) Schematic graph representation for Example 5.

Domain consistency and propagation algorithm

A feasible flow on the introduced graph representation reflects a feasible assignment of the `nested_gcc` constraint. A flow going from a variable vertex x_i^k to a value vertex d^k corresponds to the assignment $x_i^k = d$. In addition a value vertex d^k collects all the flow coming from duplicate value vertices $d^j, j \geq k$, which means it receives a flow equal to the number of assignments of d to variables in $\mathbb{X}^k$. For a given value vertex, the bounds on the single outgoing arc constrain the number of occurrences according to the definition of the constraint, by construction.

Theorem 12. *There is a bijection between solutions to the* `nested_gcc` *and feasible flows in the related graph representation G.*

Proof. $\Rightarrow$ Given a solution, we can build a feasible flow setting a unitary flow in the arc (x_i^k, d_j^k) for each assignment $x_i^k = d_j$. The arcs in A_s are all saturated. An arc $(d_j^k, v) \in A_{req}^k$ has a flow equal to $\#(d_j, \tau \downarrow_{\mathbb{X}^k})$. Note that for any given k and vertex d_j^k, demands and capacities of the arc $(d_j^k, v) \in A_{req}^k$ are satisfied since the related flow is equivalent to the sum of the flow coming from level k and higher.
$\Leftarrow$ Given a feasible (integral) flow, we build an assignment setting $x_i^k = d_j$ whenever $f(x_i^k, d_j^k) = 1$. □

Consider again Example 5: the constraint is infeasible and there is no feasible flow in the related graph G of Figure 2.3.

Corollary 1. *Let G be the graph representation of a* `nested_gcc` *and f a feasible flow on G. The constraint is domain consistent iff for each arc $a \in A_{X^k}$ there exists a feasible flow such that $f(a) = 1$.*

Proof. From Theorem 12, if there exists a feasible flow that has $f(a) = 1$ with $a = (x_i^k, d^k)$ then there exists a solution with $x_i^k = d$. Analogously, if there exists a solution with $x_i^k = d$ then there exists a flow with $f(a) = 1$ where $a = (x_i^k, d^k)$. □

Following Régin in [89], we can design a filtering algorithm in which we find a feasible flow in the graph representation in order to check the feasibility of the constraint. If it does not exist then the constraint is infeasible. Otherwise, we compute the strongly connected component [103] of the residual graph and then every arc that does not belong to any strongly connected component can be removed.

Complexity In the following, we use N to indicate the total number of variables and d for $|D_{\mathbb{X}}|$. The propagation of the `nested_gcc` requires $O(nm)$ time to find a feasible flow (Ford-Fulkerson) and $O(n+m)$ to find infeasible values where n is the number of vertices and m is the number of arcs of the `nested_gcc` graph representation. Here, n is in $O(N+d\ell)$ and m is in $O(Nd+d\ell)$. Note that the equivalent set of `gcc`'s representing the `nested_gcc` requires the propagation of ℓ different `gcc`'s. A single `gcc` propagates in $O(\sqrt{n'}m')$ [83] where n' and m' are respectively the number of vertices and the number of arcs of the `gcc` graph representation and $n' \in O(N+d)$ and $m' \in O(Nd)$.

2.2.2 Further generalization

So far, skill levels have been considered linearly ordered: a resource of level k is able to satisfy requirements of levels $k, k-1, \ldots, 1$. The main challenge is now how far we can generalize relations between skill levels in order to address more complex problems while still using the flow algorithm.

Different skill level relations are shown in Figure 2.4: in (a) the skill levels are linearly ordered while in (b) levels are organized in a tree-like fashion. The semantics of Figure 2.4(b) is that both resources of type β and γ can accomplish a task with requirements of type α, whereas resources of type β cannot satisfy requirements of type γ (and the other way around). Equivalently, we write $\delta \succ \beta$, $\beta \succ \alpha$, $\epsilon \succ \gamma$, $\zeta \succ \gamma$, $\eta \succ \gamma$ and $\gamma \succ \alpha$, where we consider $\succ$ a reflexive, antisymmetric and non-transitive relation. The transitive closure of $\succ$ is denoted by $\succ^*$ (hence, for instance $\delta \succ^* \alpha$). Note that the relation between resource classes is not a partial order relation: we cannot have $\lambda \succ \mu$ and $\lambda \succ \nu$ or, in other words, lower classes cannot rejoin in a single higher class. The reason of this limitation will be clarified at the end of this subsection. Furthermore the relation set is such that there exists only a single root: a definition of multiple roots (a forest) simply gives rise to different constraints.

Example 6. *A company is planning to develop two software components c_1 and c_2 for an application. The component c_1 requires between 7 and 10 programmers while c_2 between 8 and 10. Particularly, both components need 1 or 2 expert developers and 3 or 4 testers. A programmer is either a basic developer or an expert developer or a tester, however both expert developers and testers can accomplish duties as a basic developer ("expert developer" $\succ$ "basic developer", "tester" $\succ$ "basic developer"). The company has 4 novices, 8 testers and 3 expert developers. The different relations and component requirements are depicted in Figure 2.5. A possible solution is to assign 4 testers for each component; one tester for each component should work as a basic developer. Novices are evenly divided between the two tasks, one expert developer will be assigned to the development of component c_1 and finally the remaining two expert developers will work for the component c_2 (one as a basic developer).*

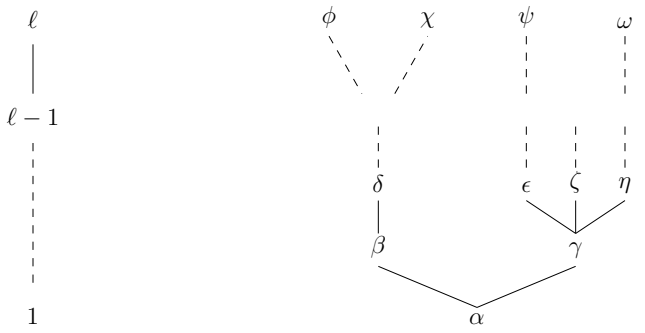


Figure 2.4 Skill level relations: (a) linearly ordered skill levels and (b) tree-like ordered skill levels.

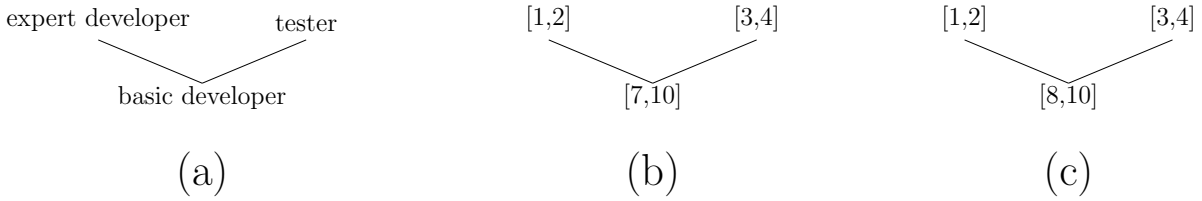


Figure 2.5 (a) Programmers skill relations. (b) Requirements for component c_1. (c) Requirements for component c_2.

Note that, more generally, whenever we have a taxonomy or hierarchy of resources, we can easily derive the resource relations. This scenario fits perfectly applications in which resources are represented as classes in a UML class diagram and they are organized in a hierarchy (with single inheritance); a subclass by definition is a specialization of the superclass, it is able to act as the superclass (the subclass "is" a kind of superclass) but it has additional capabilities.

We now formally introduce the constraint that models the described problem substructure. In the following, Σ represents the set of different resource classes where, arbitrarily, α is considered the lower level (i.e. the root class). The variables representing resources of class $\gamma \in \Sigma$ are denoted by X^γ. We write $\mathbb{X}^\lambda = \bigcup\{X^\gamma \,|\gamma \in \Sigma, \gamma \succ^* \lambda\}$ to represents the union of the variables of level λ and higher w.r.t. the relation $\succ$ [2]. For short, we write $\mathbb{X} = \mathbb{X}^\alpha$.

Definition 15 (hierarchical gcc)**.** *The hierarchical global cardinality constraint is formally defined as*

$$T(\texttt{hierarchical\_gcc}(\texttt{X}^\alpha, \ldots, \texttt{X}^\omega, (\texttt{l}^\alpha, \texttt{u}^\alpha), \ldots, (\texttt{l}^\omega, \texttt{u}^\omega), \succ)) =$$

$$\{\tau = (d_1^\alpha, \ldots, d_{n_\alpha}^\alpha, d_1^\beta, \ldots, d_{n_\omega}^\omega) \mid d_i^\gamma \in D_i^\gamma, \forall \gamma \in \Sigma, \forall d \in D_\mathbb{X} : l_d^\gamma \leq \#(d, \tau \downarrow_{\mathbb{X}^\gamma}) \leq u_d^\gamma\}$$

2. of which the linear order relation used for the `nested_gcc` is a special case

Graph Representation

The graph representation is similar to the one introduced for the `nested_gcc`; it differs mainly in how the `gcc` subgraphs are connected. We have a `gcc` substructure for each resource class; value vertices are still duplicated and they are connected to the equivalent value vertices following the resource relations. Note again that resources can be only of a given class, hence a resource is represented by exactly one vertex. The total amount of flow coming out from a variable vertex is still unitary.
More formally, the graph $G = (V, A)$ is defined as follows:

$$V = \mathbb{X} \cup \big(\bigcup_{\gamma \in \Sigma} D^{\gamma}_{\mathbb{X}} \big) \cup \{s, t\}$$

$$A = A_s \cup \big(\bigcup_{\gamma \in \Sigma} A_{X^{\gamma}} \big) \cup \big(\bigcup_{\gamma \in \Sigma} A^{\gamma}_{req} \big)$$

where

$$\begin{aligned}
A_s &= \{(s, x^{\gamma}_i) \mid \gamma \in \Sigma\,,\ i = 1, \ldots, n_{\gamma}\} \\
A_{X^{\gamma}} &= \{(x^{\gamma}_i, d^{\gamma}_j) \mid \gamma \in \Sigma, d^{\gamma}_j \in D^{\gamma}_i\} \\
A^{\gamma}_{req} &= \begin{cases} \{(d^{\alpha}_i, t) \mid i = 1, \ldots, |D_{\mathbb{X}}|\} & \text{if } \gamma = \alpha \\ \{(d^{\gamma}_i, d^{\lambda}_i) \mid i = 1, \ldots, |D_{\mathbb{X}}| : \gamma \succ \lambda\} & \text{if } \gamma \neq \alpha \end{cases}
\end{aligned}$$

Arcs in A_s have unitary lower and upper bounds, whereas arcs in $A_{X^{\gamma}}$ have null lower bounds and unitary upper bounds. Each arc $(d^{\gamma}_i, v) \in A^{\gamma}_{req}$ has lower and upper bound respectively equal to $l^{\gamma}_{d_i}$ and $u^{\gamma}_{d_i}$.

An example is given in Figure 2.6; four classes of resources are defined with the following relations: $\delta \succ \beta$, $\beta \succ \alpha$ and $\gamma \succ \alpha$. The equivalent constraint graph representation is shown in Figure 2.6b.

The technical limitation why it is not possible to express resource relations as a partially ordered set (poset) is that we might have a situation like: $\lambda \succ \mu$ and $\lambda \succ \nu$. Thus, the outcome of a `gcc` substructure may have to flow in two different `gcc` substructures: $\texttt{gcc}^{\lambda}$ should output both in $\texttt{gcc}^{\mu}$ and $\texttt{gcc}^{\nu}$. Doubling the input flow (and consequently the output flow) of the $\texttt{gcc}^{\lambda}$ will clearly lead to inconsistencies inside the `gcc` substructure.

Domain consistency and propagation algorithm

Theorem 13. *There is a bijection between solutions to the* `hierarchical_gcc` *and feasible flows in the related graph representation G.*

Proof. $\Rightarrow$ Given a solution, we can build a feasible flow setting a unitary flow in the arc $(x^{\gamma}_i, d^{\gamma}_j)$ for each assignment $x^{\gamma}_i = d_j$. The arcs in A_s are all saturated. An arc $(d^{\gamma}_j, v) \in A^{\gamma}_{req}$

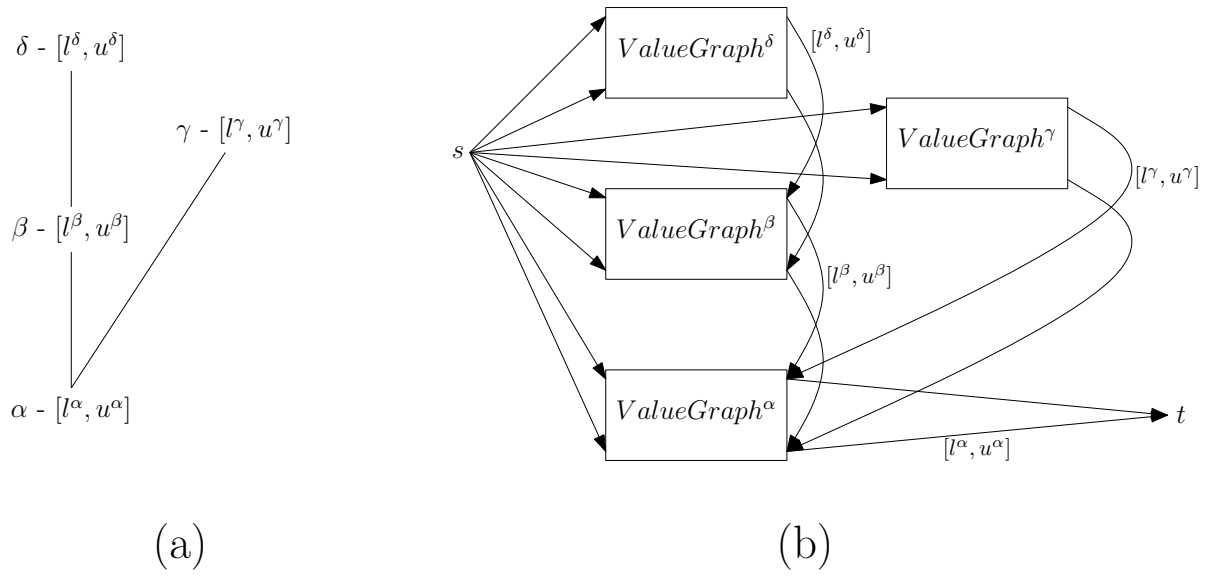


Figure 2.6 (a) Resource relation. (b) Constraint graph representation.

has a flow equal to $\#(d_j, \tau \downarrow_{\mathbb{X}^\gamma})$.
$\Leftarrow$ Given a feasible (integral) flow, we build an assignment setting $x_i^\gamma = d_j$ whenever $f(x_i^\gamma, d_j^\gamma) = 1$. □

Corollary 2. *Let G be the graph representation of a* `hierarchical_gcc` *and f a feasible flow on G. The constraint is domain consistent iff for each arc $a \in A_{X^\gamma}$ there exists a feasible flow such that $f(a) = 1$.*

Proof. From Theorem 13, if there exists a feasible flow that has $f(a) = 1$ with $a = (x_i^\gamma, d_j^\gamma)$ then there exists a solution with $x_i^\gamma = d_j$. Analogously, if there exists a solution with $x_i^\gamma = d_j$ then there exists a flow with $f(a) = 1$ where $a = (x_i^\gamma, d_j^\gamma)$. □

The propagation algorithm works exactly as in the `nested_gcc` with the only difference given by the graph. The resulting complexity is then equivalent, that is, $O(nm)$ where n and m are respectively the number of vertices and edges of the graph representation. Here, n is in $O(N + d\ell)$ and m is in $O(Nd + d\ell)$.

2.2.3 Experimental results

We implemented the `nested_gcc` and we compared them with the equivalent set of `gcc`'s. We chose as benchmark some random instances of the ROADEF challenge. We recall briefly that the problem consists of grouping technicians in teams in order to accomplish a set of tasks. A technician has skills in different domains and, particularly, he has associated a skill

level for each domain; a technician is able to satisfy requirements of his skill level and lower. A task requires a specified number of technicians for each pair domain-level in order to be accomplished. The goal is to form teams of technicians such that they are able to perform a given set of tasks.

The problem is modeled as a set of `nested_gcc`'s one for each domain where the variables represent the technicians and the values represent the tasks. As variable selection heuristic, we developed an ad-hoc heuristic that chooses the most skilled technicians first; from preliminary tests this heuristic seemed to narrow the gap between `nested_gcc` and the set of `gcc` representations. The testbed has been generated as follows: each task has associated an optimistic approximation of the technicians needed; then from the set of tasks, we chose randomly a subset such that the sum of the approximations is less than the number of available technicians. Furthermore, we randomly removed values from variable domains according to an input percentage.

The constraint has been implemented with Ilog Solver 6.2 and the tests were performed on a machine with an AMD Dual Opteron 250 (2.4GHz) with 3GB RAM (note however that only one processor has been used). We set a time limit of 600 seconds for each run. We compared two different models: the former exploits the `nested_gcc`, the latter uses traditional `gcc`'s. For a fair comparison, the second model has been solved using both our implementation of the `gcc` and ILOG's. The results are shown in Table 2.1 where columns are: instance name, percentage of domain value removals; time (in seconds) and backtracks for the `nested_gcc`; time (in seconds) for the `gcc` model and number of backtracks and finally the time of ILOG's implementation of the gcc modelling.

Results are given for individual instances in which the `gcc` model took more than 1 second whereas for the remaining 17 easy instances just the average is reported. Instances from data11 have 4 skill domains with 4 skill levels each whereas instances from data12 have 5 domains with 3 skill levels each. Each instance has been tried with different percentages of domain value removals (shown in the second column). The remaining columns show the running times (for finding a solution or proving infeasibility) and the number of backtracks respectively for the `nested_gcc`, `gcc` and ILOG's; in all three approaches the same variable and value ordering heuristics have been used. If the running time is not shown, the solver timed out either without finding a solution or without proving the infeasibility of the instance (however in those cases we show the number of backtracks performed within the time limit).

Depending on the instance, the reduction on the number of backtracks using the `nested_gcc` can go from null to two orders of magnitude; whenever the reduction is significant, we obtain better running times. Nonetheless there are instances in which even with our implementation of the `gcc` we get better performances over the `nested_gcc`. Hence, further studies are

Table 2.1 Experimental results for `nested_gcc`; time are expressed in seconds

Instance	Perc.	`nested_gcc`		`gcc`'s		
		Time	Bcks	Time	Bcks	ILOG Time
data11-a	0.3	13.61	73381	14.12	73381	**11.92**
data11-b	0.3	**3.85**	10992	4.65	17454	3.96
data11-b	0.2	**35.75**	106324	56.27	202690	51.76
data11-b	0.4	3.1	8267	2.75	9219	**2.51**
data11-b	0.5	1.76	4986	**1.36**	5382	1.52
data11-c	0.1	6.43	23531	5.49	23778	**3.52**
data11-c	0.2	3.18	11771	2.71	11979	**1.81**
data11-d	0.1	**4.15**	16478	7.04	26766	5.59
data12-a	0.3	**140.43**	422107	-	2373366	-
data12-a	0.4	**0.16**	419	135.44	505243	154.01
data12-a	0.5	**0.11**	300	6.47	22099	6.64
data12-b	0.4	**20.16**	45634	36	109762	53.24
data12-c	0.4	1.12	2835	1.55	5475	**1.02**
data12-c	0.3	**13.24**	36336	20.12	75787	14.74
data12-d	0.4	179.00	353462	**148.62**	445726	172.92
data12-d	0.5	98.02	185798	74.77	203920	**74.04**
17 inst.(<1sec)		**0.07**	230	0.13	458	0.13

required in order to better characterize the instances and understand when the use of the `nested_gcc` is likely to lead to better running times. We think that an instance generator with a more fine grained parameterization could help us in this task as well as in generating a broader testbed. In fact, the basic generator produced too many instances either too easy or too hard, the same problem that we encountered also during the generation of a testbed for the `hierarchical_gcc`.

2.2.4 Softening the Nested Global Cardinality Constraint

For overconstrained problems, it might be useful to soften the `nested_gcc`. We can think of typical scenarios in which highly-skilled resources are limited whereas low-skilled ones are numerous. Such cases are dealt in real-life by assigning a resource of a given level to a task that requires slightly higher skill-level.

We propose here a soft version of the `nested_gcc` that actually capture this concept, i.e., it allows, to some extent, low-level resources to be assigned to higher level-resources if necessary (note this softening approach applies straight-forwardly also to the `hierarchical_gcc`).

We formally define the `soft_nested_gcc` as:

$$T(\texttt{soft\_nested\_gcc}(\texttt{X}^1, \ldots, \texttt{X}^\ell, (\texttt{l}^1, \texttt{u}^1), \ldots, (\texttt{l}^\ell, \texttt{u}^\ell)), \texttt{Z}) =$$

$$\{\tau = (d_1^1, \ldots, d_{n_1}^1, d_1^2, \ldots, d_{n_\ell}^\ell) \mid d_i^k \in D_i^k, \forall 1 \leq k \leq \ell, \forall d \in D_{\mathbb{X}} : l_d^k \leq \#(d, \tau) \leq u_d^k,$$

$$violation(d_1^1, \ldots, d_{n_1}^1, d_1^2, \ldots, d_{n_\ell}^\ell) \leq d_z\}$$

where the violation is defined as:

$$violation(d_1^1, \ldots, d_{n_1}^1, d_1^2, \ldots, d_{n_\ell}^\ell) = \sum_{d \in D_{\mathbb{X}}} \sum_{1 < k \leq \ell} max(l_d^k - \#(d, \tau \downarrow_{\mathbb{X}^k}), 0)$$

As an example, consider a single task whose lower bounds for three different skill-level are $l_d^3 = 3$, $l_d^2 = 3$ and $l_d^1 = 3$; suppose furthermore that only three resources are available, one for each skill-level, that is $X^3 = \{x_1^3\}$, $X^2 = \{x_1^2\}$, $X^1 = \{x_1^1\}$. To satisfy the constraint all three resources must be assigned to the task; particularly, x_1^2 will cause a violation equal to 1 and x_1^1 will cause a violation equal to 2 as it assigned to a task that is two levels higher than its level; x_1^3 does not cause any violation.

The filtering algorithm for such constraint is inspired directly by Van Hoeve et al. in [107]; in that approach, arcs representing the violation are introduced to the original variable-value graph; these additional arcs have a strictly positive cost and are used iff necessary. The filtering algorithm is based on minimum-cost maximum-flow algorithm (see [107] or [89] for further details).

The graph representation for the `soft_nested_gcc` introduces for each level $k < \ell$ upward violation arcs going from the value-vertices of level k to the value vertices of level $k + 1$. Formally, the directed graph $G = (V, A)$ is defined as follows:

$$V = \mathbb{X} \cup \big(\bigcup_{k=1}^{\ell} D_{\mathbb{X}}^k\big) \cup \{s, t\}$$

$$A = A_s \cup \big(\bigcup_{k=1}^{\ell} A_{X^k}\big) \cup \big(\bigcup_{k=1}^{\ell} A_{req}^k\big)$$

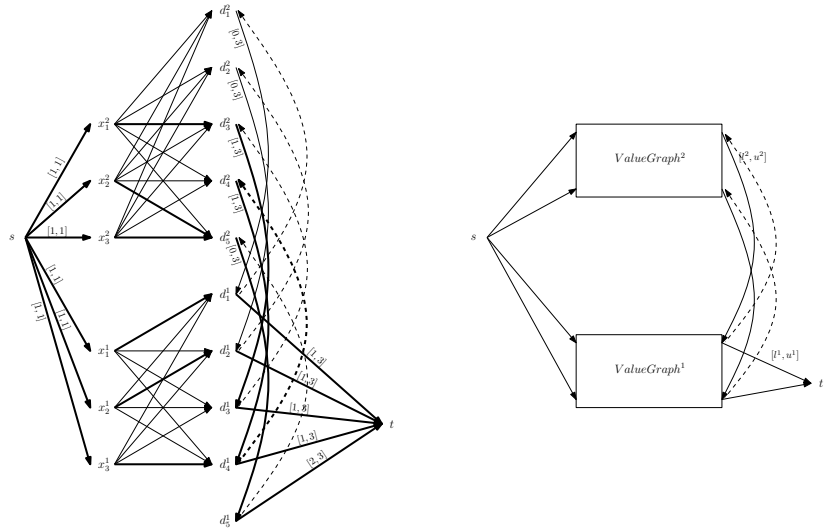
where

$$\begin{array}{lcl} A_s & = & \{(s, x_i^k) \mid k = 1, \ldots, \ell\ ,\ i = 1, \ldots, n_k\} \\ A_{X^k} & = & \{(x_i^k, d_j^k) \mid i = 1, \ldots, n_k, d_j^k \in D_i^k\} \\ A_{req}^k & = & \begin{cases} \{(d_i^1, t) \mid i = 1, \ldots, |D_{\mathbb{X}}|\} & \text{if } k = 1 \\ \{(d_i^k, d_i^{k-1}) \mid i = 1, \ldots, |D_{\mathbb{X}}|\} & \text{if } 2 \leq k \leq \ell \end{cases} \\ A_{upwards}^k & = & \{(d_i^k, d_i^{k+1}) \mid i = 1, \ldots, |D_{\mathbb{X}}|\} \quad \text{if } 1 \leq k < \ell \end{array}$$

Arcs belonging to set $A_{upwards}^k$ have infinite capacity and a unitary cost whereas the others have a cost equal to zero. A minimum-cost flow will minimize the use of arcs with positive costs therefore finding a solution with no violation if exists. The total violation total corresponds to the cost of the flow, and the constraint fails iff that cost is higher than the current upper bound of the violation variable Z; domain filtering can be performed based on

cost reasoning as in [107].

In Figure 2.7, we show the graph representation of the `soft_nested_gcc` for Example 5; that instance of `nested_gcc` is infeasible. However, if we soften the constraint we can find a solution with minimum violation as shown in Figure 2.7 (bold arcs are employed in a possible min-cost maximum flow). One possible solution with unitary violation assign the variables as follows: $x_1^2 = d_3$, $x_2^2 = d_5$, $x_3^2 = d_5$, $x_1^1 = d_1$, $x_2^1 = d_2$, $x_3^1 = d_4$. Note, that the lower bounds for what concerns skill level one are satisfied: all the tasks have one resource assigned and task d_5 has two. In respect of skill level two, d_3 has one level-2 resource assigned (x_1^2) nonetheless d_4 has to be performed by a resource of lower level (x_3^1) that is from where comes the violation.



(a) (b)

Figure 2.7 (a) `soft_nested_gcc` Graph Representation for Example 5: dashed arcs have unitary cost and bold arcs are used by one possible min-cost maximum flow. (b) Schematic graph representation for Example 5.

Note that the granularity of the violation can be pushed even further as in [75] where violation arcs may have associated different costs (this kind of violation measures are referred to as σ-violation). This more fine-grained distribution of costs allows to represent preferences on the values (tasks) to satisfy, that is, high priority values have high-cost incident arcs.

2.2.5 Expressing preferences

For ease of presentation, in this section we take into consideration only linearly ordered resources. However the results can be easily extended to the hierarchical version.

In the constraints presented, there might be cases in which we would like to define some preferences among different consistent assignments. Imagine, for example, that a given task requires one resource of level 1 and one of level 2; however only one resource of level 2 and one of level 3 are available. The constraint does not allow to express a difference between a solution in which the level 3 resource will perform level 1 duties or a solution in which the level 2 resource will carry out level 1 duties and the level 3 resource level 2 duties. Nonetheless, in both solutions we simply have the two resources assigned to the same task without any information about who is going to perform what. We should then enrich the model in order to express this new information. In this new setting, domains should contain for each task different values to denote different duty levels. So, d_j^k represents the task j with duty level k; assignment $x_i^{k'} = d_j^k$ means that resource i of level k' is going to perform duties of level k of task j. For the sake of clarity, note that in the `nested_gcc` we would have had simply $x_i^{k'} = d_j$; the differentiation of duty levels is only present inside the graph representation but not at the constraint level; for expressing preferences such differentiation needs to be brought up to the constraint level.

In the graph representation of `nested_gcc` with preferences, a variable $x_i^{k'}$ is connected directly to value vertices d_j^k with $k \leq k'$. Lower and upper bounds of value occurrences are expressed directly for each d_j^k. It follows that occurrences of a value d_j^k do not interfere with the ones of value $d_j^{k'}$ with $k \neq k'$. Hence, the graph representation does not contain anymore the arcs A_{req}^k that connect value vertices of different levels but rather value vertices are directly connected to the sink. In order to express preferences, we should also introduce positive costs on the arcs $(x_i^{k'}, d_i^k)$ whenever $k < k'$. Thus, the overall graph representation is similar to a particular case of the $\texttt{cost} - \texttt{gcc}$ [90]. Figure 2.8 shows an example of a `nested_gcc` with preferences in which we have 5 variables and 9 values (5 resources with 3 different skill levels and 3 tasks leading to an overall of 9 different values). With such a representation it is also straightforward to constrain the eventual gap in a given assignment between the resource level and the duty level.

Note that this particular graph representation could not be used for the `nested_gcc`. For instance, take in consideration Figure 2.8: if this would have been a traditional `nested_gcc`, value vertices represent simply tasks, hence d_2^2 denotes the task 2 as well as d_2^1. Suppose furthermore that task 2 requires at most 1 resource of level 2 or higher. The constraint would be consistent even with assignments $x_1^2 = d_2^1$ and $x_2^2 = d_2^2$ hence leading to a contradiction.

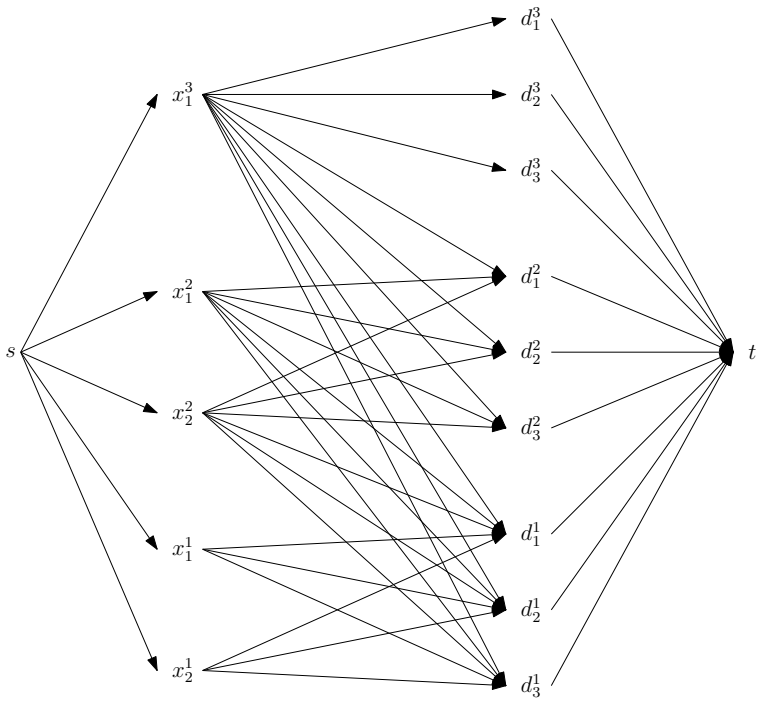


Figure 2.8 Graph representation for the nested gcc with preferences.

2.3 Summary

In Section 2.1, we have presented two algorithms for reaching generalized arc consistency in the Soft Global Cardinality Constraint with variable-based violation and value-based violation. They check the consistency of the constraint with a running time complexity of $O(\sqrt{n}m)$ and they prune inconsistent values in $O(m+n)$ where n is the cardinality of the set of variables and $m = \sum_i |D_i|$. We outperform previous algorithms that ran in $O(n(m + n\log n))$ (variable-based violation) and $O((n+k)(m + n\log n))$ (value-based violation) for constraint consistency check and in $O(\Delta(m + n\log n))$ for domain pruning where $\Delta = min(n, k)$ $(k = |\bigcup_i D_i|)$.

In Section 2.2, we proposed generalizations of the `gcc` to address certain resource allocation problems. We showed both theoretically and empirically that such generalizations can outperform an alternate formulation using `gcc`'s.

CHAPTER 3

Counting Algorithms

In this chapter, we will present some algorithms to count the number of solutions of some constraints (portions of this chapter appeared in [113], [114] and [116]). Particularly, we will tackle two important families of constraints: occurrence counting constraints, such as `alldifferent` and its extension `global_cardinality_constraint`, and sequencing constraints, such as `regular`.

In a simplified view, two of the main methods that constraints expose to solvers are:

– `Constraint` :: `isFeasible`()

– `Constraint` :: `isConsistent`(`Variable`, `Value`)

The former allows to verify whether a constraint is feasible i.e. there exists at least a combination of values assigned to the variables in the constraint scope that satisfies the constraint. The latter is to verify whether a variable-value pair belongs to at least one of such combination to possibly filter it. Sometimes, these two methods may be combined into a single method that verifies the feasibility of the constraint and consistency of each variable-value pair.

What we propose is to enrich the constraint interface with two additional methods:

– `Constraint` :: `getSolutionCount`()

– `Constraint` :: `getSolutionDensity`(`Variable`, `Value`)

They rely on the following concepts:

Definition 16 (Solution Count)**.** *Given a constraint $\gamma(x_1, \ldots, x_k)$ and respective finite domains $D_{i\ 1\leq i\leq k}$, let $\#\gamma(x_1, \ldots, x_k)$ denote the number of solutions of constraint γ.*

Definition 17 (Solution Density)**.** *Given a constraint $\gamma(x_1, \ldots, x_k)$, respective finite domains $D_{i\ 1\leq i\leq k}$, a variable x_i in the scope of γ, and a value $d \in D_i$, we will call*

$$\sigma(x_i, d, \gamma) = \frac{\#\gamma(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_k)}{\#\gamma(x_1, \ldots, x_k)}$$

the solution density [1] *of pair (x_i, d) in γ. It measures how often a certain assignment is part of a solution to the constraint.*

The chapter is organized as follows: in Section 3.1 we will propose and analyze some counting algorithms for the `alldifferent` constraint. In Section 3.2, we extend the results

1. Also referred to as the *marginal* in some of the literature.

to the `gcc` constraint. Section 3.3 introduces a counting algorithm for the `regular` constraint. Finally, a summary will be given in Section 3.4.

3.1 Counting for alldifferent Constraints

The `alldifferent` constraint restricts a set of variables to be pairwise different [88].

We recall here the formal definition:

Definition 18. *Given a set of variables $X = \{x_1, \ldots, x_n\}$ with respective domains $D_1, \ldots, D_n$, we define the* `alldifferent` *as:*

$$T(\texttt{alldifferent}(X)) = \{(d_1, d_2, \ldots, d_n) \mid d_i \in D_i, d_i \neq d_j \ \ for\ i \neq j\}$$

Definition 19 (Value Graph [88])**.** *Given a set of variables $X = \{x_1, \ldots, x_n\}$ with respective domains $D_1, \ldots, D_n$, we define the* value graph *as a bipartite graph $G = (X \cup D_X, E)$ where $D_X = \bigcup_{i=1,\ldots,n} D_i$ and $E = \{\{x_i, d\} \mid d \in D_i\}$.*

There exists a bijection between a maximum matching of size $|X|$ on the value graph and a solution of the related `alldifferent` constraint (see [88]). Finding the number of solutions is then equivalent to counting the number of maximum matchings on the value graph.
Maximum matching counting is also equivalent to the problem of computing the permanent of a (0-1) matrix. Given a bipartite graph $G = (V_1 \cup V_2, E)$, with $|V_1| = |V_2| = n$, the related $n \times n$ adjacency matrix A has element $a_{i,j}$ equal to 1 if and only if vertex $i \in V_1$ is connected to vertex $j \in V_2$.

Note that the matrix is not symmetric, rows represents the X, columns D_X.

The permanent of a $n \times n$ matrix A is formally defined as:

$$per(A) = \sum_{\sigma \in S_n} \prod_i a_{i,\sigma(i)} \tag{3.1}$$

where S_n denotes the symmetric group, i.e. the set of $n!$ permutations of $[n]$. Given a specific permutation, the product is equal to 1 if and only if all the elements are equal to 1 i.e. the permutation is a valid maximum matching in the related bipartite graph. Hence, the sum over all the permutations gives us the total number of maximum matchings.
Equivalently, the permanent can be expressed following Laplace's expansion formula:

$$per(A) = \sum_{j=1}^{n} a_{1,j}\, per(A_{1,j}) \tag{3.2}$$

where $A_{1,j}$ denotes the submatrix obtained from A by removing row 1 and column j (the permanent of the empty matrix is equal to 1). In the following, we will freely use both matrix and graph representations.

Note that the permanent is defined on square matrices i.e. the related bipartite graph needs to have $|V_1| = |V_2|$. In order to overcome this limitation, we can augment the graph by adding $|V_2| - |V_1|$ fake vertices to V_1 (without loss of generality $|V_1| \leq |V_2|$) each one connected to all vertices in V_2.

Theorem 14. *Let $G(V_1 \cup V_2, E)$ be a bipartite graph with $|V_1| \leq |V_2|$ and the related augmented graph $G'(V_1' \cup V_2, E')$ a graph such that $V_1' = V_1 \cup V_{fake}$ with $|V_{fake}| = |V_2| - |V_1|$ and the edge set $E' = E \cup E_{fake}$ with $E_{fake} = \{(v_i, v_j) \mid v_i \in V_{fake}, \quad v_j \in V_2\}$. Let $|M_G|$ and $|M_{G'}|$ be the number of maximum matchings respectively in G and G'. Then $|M_G| = |M_{G'}|/|V_{fake}|!$.*

Proof. Given a maximum matching $m \in M_G$ of size $|V_1|$, since m covers all the vertices in V_1 then there exists exactly $|V_2| - |V_1|$ vertices in V_2 not matched. In the corresponding matching (possibly not maximum) $m' = m$ in G', the vertices in V_2 that are not matched can be matched with any of the vertices in V_{fake}. Since each vertex in V_{fake} is connected to any vertex in V_2 then there exists exactly $|V_{fake}|!$ permutations to obtain a perfect matching in G' starting from a maximum matching m in G. If there is no maximum matching of size $|V_1|$ for G then clearly there isn't any of size $|V_2|$ for G' either. □

For simplicity in the rest of the section we assume $|X| = |D_X|$.

3.1.1 Computing the Permanent

The problem of computing the permanent has been studied for the last two centuries and it is still a challenging problem to address. Even though the analytic formulation of the permanent (Formula 3.2) resembles that of the determinant, there have been few advances on its exact computation. In 1961 [62], Kasteleyn solved the problem only for a particular class of matrices (the one that represents Pfaffian graphs which is a superset of planar graphs); his algorithm runs in $O(n^3)$.

In 1963, Ryser [50] proposed an exact algorithm for the general case whose running time is $\Theta(n2^n)$, thus it is not practical for our purposes. In 1979, Valiant [105] proved that the problem is $\#P$-complete [2], even for 0-1 matrices, that is, under reasonable assumptions, it cannot be computed in polynomial time in the general case. The focus then moved to approximating the permanent. We can identify at least four different approaches for approximating the

2. Note however that enforcing arc, bounds or domain consistency on the `alldifferent` constraint restricts the class of 0-1 matrices for which we need to compute the permanent. We believe however that the problem remains #P-complete

permanent: elementary iterative algorithms, reductions to determinants, iterative balancing, and Markov Chain Monte Carlo methods.

Elementary Iterative Algorithms. Rasmussen proposed in [85] a very simple recursive estimator for the permanent. This method works quite well for dense matrices but it breaks down when applied to sparse matrices; its time complexity is $O(n^3\omega)$ recently improved to $O(n^2\omega)$ by Fürer et al. [32] (here ω is a function satisfying $\omega \to \infty$ as $n \to \infty$). Further details about these approaches will be given in the next section.

Reduction to Determinant. The determinant reduction technique, firstly proposed by Godsil and Gutnam [38], is based on the resemblance of the permanent and the determinant. This method randomly replaces some 1-entry elements of the matrix by uniform random elements $\{\pm 1\}$. It turns out that the determinant of the new matrix is an unbiased estimator of the permanent of the original matrix. The proposed algorithms either provide an arbitrarily close approximation in exponential time [20] or an approximation within an exponential factor in polytime [6].

Iterative Balancing. The work of Linial et al. [69] exploits a lower bound on the permanent of a doubly stochastic[3] $n \times n$ matrix B: $per(B) \geq n!/n^n$. The basic idea is to use the linearity of permanents w.r.t. multiplication with constants and transform the original matrix A to an approximated doubly stochastic matrix B and then exploit the lower bound. The algorithm that they proposed runs in $O(n^5 \log^2 n)$ and gives an approximation within a factor of e^n.

Markov Chain Monte Carlo Methods. Markov Chains can be a powerful tool to generate almost uniform samples. They have been used for the permanent in [18] and [55] but they impose strong restrictions on the minimum vertex degree. A remarkable breakthrough was achieved by Jerrum et al. [57]: they proposed the first polynomial approximation algorithm for general matrices with non-negative entries. This fully polynomial randomized approximation scheme (FPRAS) can be made arbitrarily close to the exact value of the permanent. Nonetheless this remarkable result has to face its impracticality due to a very high-computational complexity $\tilde{O}(n^{26})$ improved to $\Theta(n^{10} \log^3 n)$ later on [58].

Note that for our purposes we are not only interested in computing the total number of solutions but we also need solution densities for each variable-value pair. Moreover, we need

3. $\sum_i a_{i,j} = \sum_j a_{i,j} = 1$

fast algorithms that work for most matrices; since the objective is to build a search heuristic based on counting information, we would prefer a fast algorithm with less precise approximation over a slower algorithm with better approximation guarantees. With that in mind, Markov Chain-based algorithms do not fit our needs (they are either too slow or they have a precondition on the minimum vertex degree). Algorithms based on determinants or matrix scaling are either exponential in time or give approximations that are too loose (within an exponential factor). The approach that seems to suit our needs better is elementary iterative algorithms. It combines a reasonable complexity with a good approximation. Although it gives poor results for sparse matrices, those cases are likely to appear close to the leaves [4] of the search tree where an error by the heuristics has a limited negative impact.

3.1.2 Rasmussen's Estimator and Its Extensions

Suppose we want to estimate a function Q (in our case the permanent): a traditional approach is to design an estimator that outputs a random variable X whose expected value is equal to Q. The estimator is unbiased if $E(X)$ and $E(X^2)$ are finite. A straightforward application of Chebyshev's inequality shows that if we conduct $O(\frac{E(X^2)}{E(X)^2}\epsilon^{-2})$ independent and identically distributed trials and we take the mean of the outcomes then we are guaranteed an ϵ-approximation. Hence the performance of a single run of the estimator and the ratio $\frac{E(X^2)}{E(X)^2}$ (called the *critical ratio*) determine the efficiency of the algorithm.

We briefly recall here that a *randomized approximation scheme* for a counting problem is a randomized algorithm that takes as input an instance x of a decision problem, and outputs a random variable $N \in \mathbb{N}$ representing the number of solutions of the decision problem that has "high probability" [5] of being close to the exact value; formally:

$$P[e^{-\epsilon} f(x) \leq N \leq e^{\epsilon} f(x)] \geq 3/4$$

where ϵ denotes the error tolerance and $f(x)$ the exact solution count. A *fully polynomial randomized approximation scheme* (fpras) is an algorithm with the above property that runs in time bounded by a polynomial in the instance size and ϵ^{-1}. The design of an fpras for approximate counting can be solved if we can generate in polytime almost uniform samples for the related decision problem [59]. For further details about fpras for counting problems see [71].

In the following, we denote by $A(n, p)$ the class of random (0-1) $n \times n$ matrices in which each element has independent probability p of being 1. We write X_A for the random variable

4. where propagation is likely to have already significantly shrunk the variable domains

5. in the literature, "high probability" usually means greater than 3/4, however the definition is robust to variations of it

that estimates the permanent of matrix A; $A_{i,j}$ denotes the submatrix obtained from A by removing row i and column j. The pseudo-code of Rasmussen's estimator is shown in Algorithm 1; the algorithm computes X_A for a matrix A and recursively calls the computation of $X_{A_{1,j}}$ on a smaller matrix $A_{1,j}$. Despite its simplicity compared to other techniques, the estimator is unbiased and shows good experimental behaviour. Rasmussen gave theoretical results for his algorithm applied to random matrices belonging to the class $A(n, p \geq 1/2)$. He proved that for "almost all" matrices of this class, the critical ratio is in $O(n\omega)$ where ω is a function satisfying $\omega \to \infty$ as $n \to \infty$; the complexity of a single run of the estimator is in $O(n^2)$, hence the total complexity is in $O(n^3\omega)$. Here "almost all" means that the algorithm gives a correct approximation with probability that goes to 1 as $n \to \infty$. While this result holds for dense matrices, it breaks down for sparse matrices. Note however that there are still matrices belonging to $A(n, p = 1/2)$ for which the critical ratio is exponential. Consider for instance the upper triangular matrix:

$$\mathbf{U} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ & 1 & \dots & 1 \\ & & \ddots & \vdots \\ & & & 1 \end{pmatrix}$$

For this particular matrix Rasmussen's estimator has expected value $E(X_U) = 1$ and $E(X_U^2) = n!$, hence the approximation is likely to be very poor.

```
1  if n = 0 then
2      X_A = 1
3  else
4      W = {j : a_{1,j} = 1};
5      if W = ∅ then
6          X_A = 0;
7      else
8          Choose j uniformly at random from W;
9          Compute X_{A_{1,j}};
10         X_A = |W| · X_{A_{1,j}};
```

Algorithm 1: Rasmussen's estimator

Fürer et al. [32] enhanced Rasmussen's algorithm with some branching strategies in order to pick up more samples in the critical parts of the matrix (Algorithm 2). It resembles very closely the exploration of a search tree. Instead of choosing u.a.r. a single column j from

W, Fürer picks up a subset $J \subseteq W$ and it iterates on each element of J. The number of times it branches is logarithmic in the size of the matrix, and for a given branching factor he showed that a single run of the estimator still takes $O(n^2)$ time. In Algorithm 2, branching points are defined by s^i and the branching factor by r; Fürer sets $s = 2$ and $r = 3$ in his proof (we kept the same values for our study). The advantage of this approach resides in the theoretical convergence guarantee: the number of required samples is only $O(\omega)$ instead of Rasmussen's $O(n\omega)$, thus the overall complexity is $O(n^2\omega)$.

```
1  if n = 0 then
2      X_A = 1
3  else
4      W = {j : a_{1,j} = 1};
5      if W = ∅ then
6          X_A = 0;
7      else
8          if n = s^i, i ≥ 1 then
9              K = r;
10         else
11             K = 1;
12         for ℓ = 1 to K do
13             Choose j_ℓ uniformly at random from W;
14             Compute X_{A_{1,j_ℓ}};
15         X_A = |W|(1/K Σ_{ℓ=1}^{K} X_{A_{1,j_ℓ}});
```

Algorithm 2: Fürer's estimator

Both Fürer and Rasmussen estimators allow to approximately compute the total number of solutions of an `alldifferent` constraint. However if we need to compute the solution density $\sigma(x_i, d, \gamma)$ we are forced to recall the estimators on the submatrix $A_{i,d}$. Hence the approximated solution density is:

$$\sigma(x_i, d, \gamma) \approx \frac{E(X_{A_{i,d}})}{E(X_A)} \tag{3.3}$$

Adding Propagation to the Estimator

A simple way to improve the quality of the approximation is to add propagation to Rasmussen's estimator. After randomly choosing a row i and a column j, we can propagate on the submatrix $A_{i,j}$ in order to remove all the 1-entries (edges) that do not belong to any maximum matching (the pseudo-code is shown in Algorithm 3). This broadens the applicability of the method; in matrices such as the upper triangular matrix, the propagation

can easily lead to the identity matrix for which the estimator performs exactly. However, as a drawback, the propagation takes an initial precomputation of $O(\sqrt{n}m)$ plus an additional $O(n+m)$ each time it is called [88] (here m is the number of ones of the matrix i.e. edges of the graph). A single run of the estimator requires n propagation calls, hence the time complexity is $O(nm)$; the overall time complexity is then $O(n^2 m\omega)$.

```
1 if n = 0 then
2     X_A = 1
3 else
4     Choose i u.a.r. from {1 ... n};
5     W = {j : a_{i,j} = 1};
6     Choose j uniformly at random from W;
7     Propagation on A_{i,j};
8     Compute X_{A_{i,j}};
9     X_A = |W| · X_{A_{i,j}};
```

Algorithm 3: Estimator with propagation

A particularity of the new estimator is that it removes a priori all the 1-entries that do not lead to a solution. Hence it always samples feasible solutions whereas Rasmussen's ends up with infeasible solutions whenever it reaches a case in which $W = \emptyset$. This opens the door also to an alternative evaluation of the solution densities; given the set of solution samples S, we denote by $S_{x_i,d} \subseteq S$ the subset of samples in which $x_i = d$. The solution densities are approximated as:

$$\sigma(x_i, d, \gamma) \approx \frac{|S_{x_i,d}|}{|S|} \tag{3.4}$$

Experimental results showed a much better approximation quality for the computation of the solution densities using samples (3.4) instead of using submatrix counting (3.3). It is worth pointing out that Fürer's provides several samples in a single run but highly biased from the decisions taken close to the root of the search tree; thus it cannot be used to compute solution densities from samples. Due to the better results obtained using samples, we decided not to apply propagation methods to Fürer's.

3.1.3 Upper Bounds

We also explored a different approach, trading some of the accuracy for a significant speedup in the counting procedure, in order to provide an algorithm that performs well on easy instances (i.e. fast enough) while keeping the lead in solving hard ones (i.e. with a reasonable estimation accuracy).

In 1963, Minc [76] conjectured that the permanent can be bounded from above by the following formula:

$$perm(A) \leq \prod_{i=1}^{n} (r_i!)^{1/r_i}. \tag{3.5}$$

where r_i is the number of one's in the i-th row.

Proved only in 1973 by Brégman [16], it has been considered for decades the best upper bound for the permanent. Recently, Liang and Bai [68], inspired by Rasmussen's work, proposed a new upper bound (with $q_i = min\{\lceil \frac{r_i+1}{2} \rceil, \lceil \frac{i}{2} \rceil\}$):

$$perm(A)^2 \leq \prod_{i=1}^{n} q_i(r_i - q_i + 1). \tag{3.6}$$

None of the two upper bounds strictly dominates the other. In the following we denote by $UB^{BM}(A)$ the Brégman-Minc upper bound and by $UB^{LB}(A)$ the Liang-Bai upper bound. Jurkat and Ryser proposed in [60] another bound:

$$perm(A) \leq \prod_{i=1}^{n} min(r_i, i).$$

However it is considered generally weaker than $UB^{BM}(A)$ (see [99] for a comprehensive literature review). Soules proposed in [98] some general sharpening techniques that can be employed on any existing permanent upper bound in order to improve them. The basic idea is to apply an appropriate combination of functions (such as row or column permutation, matrix transposition, row or column scaling) and to recompute the upper bound on the modified matrix.

Algorithm

We decided to adapt UB^{BM} and UB^{LB} in order to compute an approximation of solution densities for the `alldifferent` constraint. Recall that matrix element $a_{ij} = 1$ iff $j \in D_i$. Assigning j to variable x_i translates to replacing the i^{th} row by the unit vector $e(j)$ (i.e. setting the i^{th} row of the matrix to 0 except for the element in column j). We write $A_{x_i=j}$ to denote matrix A except that x_i is fixed to j. We call *local probe* the assignment $x_i = j$ performed to compute $A_{x_i=j}$ i.e. a temporary assignment that does not propagate to any other constraint except the one being processed.

The upper bound on the number of solutions of the `alldifferent`$(x_1, \ldots, x_n)$ constraint

with a related adjacency matrix A is then

$$\#\texttt{alldifferent}(x_1, \ldots, x_n) \leq min\{\, UB^{BM}(A),\, UB^{LB}(A)\}$$

Note that in Formula 3.5 and 3.6, the r_i are equal to $|D_i|$; since $|D_i|$ range from 0 to n, the factors can be precomputed and stored: in a vector $BMfactors[r] = (r!)^{1/r}, r = 0, \ldots, n$ for the first bound and similarly for the second one (with factors depending on both $|D_i|$ and i). Assuming that $|D_i|$ is returned in $O(1)$, computing the formulas takes $O(n)$ time. Solution densities are then approximated as

$$\sigma(x_i, j, \texttt{alldifferent}) \approx \frac{min\{\, UB^{BM}(A_{x_i=j}),\, UB^{LB}(A_{x_i=j})\}}{\eta}$$

where η is a normalizing constant.

The local probe $x_i = j$ may trigger some local propagation according to the level of consistency we want to achieve; therefore $A_{x_i=j}$ is subject to the filtering performed on the constraint being processed. Since the two bounds in Formula 3.5 and 3.6 depend on $|D_i|$, a stronger form of consistency would likely lead to more changes in the domains and on the bounds, and presumably to more accurate solution densities.

Once the upper bounds for all variable-value pairs have been computed, it is possible to further refine the solution count as follows:

$$\#\texttt{alldifferent}(x_1, \ldots, x_n) \leq \min_{x_i \in X} \sum_{j \in D_i} min\{\, UB^{BM}(A_{x_i=j}),\, UB^{LB}(A_{x_i=j})\}$$

This bound on the solution count depends on the consistency level enforced in the `alldifferent` constraint during the local probes.

If we want to compute $\sigma(x_i, j, \texttt{alldifferent})$ for all $i = 1, \ldots, n$ and for all $j \in D_i$ then a trivial implementation would compute $A_{x_i=j}$ for each variable-value pair; the total time complexity would be $O(mP + mn)$ (where m is the sum of the cardinalities of the variable domains and P the time complexity of the filtering).

Although unable to improve over the worst case complexity, in the following we propose an algorithm that performs definitely better in practice. We introduce before some additional notation: we write as D'_i the variable domains after enforcing θ-consistency [6] on that constraint alone and as $\tilde{I}$ the set of indices of the variables that were subject to a domain change due to a local probe and the ensuing filtering, that is, $i \in \tilde{I}$ iff $|D'_i| \neq |D_i|$. We describe the algorithm for the Brégman-Minc bound — it can be easily adapted for the Liang-Bai bound.

6. any consistency level achievable for the `alldifferent` constraint

The basic idea is to compute the bound for the matrix A and reuse it to speed up the computation of the bounds for $A_{x_i=j}$ for all $i = 1, \ldots, n$ and $j \in D_i$. Let

$$\gamma_k = \begin{cases} \frac{BMfactors[1]}{BMfactors[|D_k|]} & \text{if } k = i \\ \frac{BMfactors[|D'_k|]}{BMfactors[|D_k|]} & \text{if } k \in \tilde{I} \setminus \{i\} \\ 1 & \text{otherwise} \end{cases}$$

$$\begin{aligned} UB^{BM}(A_{x_i=j}) &= \prod_{k=1}^{n} BMfactors[|D'_k|] = \prod_{k=1}^{n} \gamma_k \; BMfactors[|D_k|] \\ &= UB^{BM}(A) \prod_{k=1}^{n} \gamma_k \end{aligned}$$

Note that γ_k with $k = i$ (i.e. we are computing $UB^{BM}(A_{x_i=j})$) does not depend on j; however $\tilde{I}$ does depend on j because of the domain filtering.

```
1  UB = BMbound(A) ;
2  for i = 1, ..., n do
3      varUB = UB * BMfactors[1] / BMfactors[|D_i|] ;
4      total = 0;
5      forall j ∈ D_i do
6          set x_i = j;
7          enforce θ-consistency;
8          VarValUB[i][j] = varUB;
9          forall k ∈ Ĩ \ {i} do
10             VarValUB[i][j] = VarValUB[i][j] * BMfactors[|D'_k|] / BMfactors[|D_k|];
11         total = total + VarValUB[i][j];
12         rollback x_i = j;
13     forall j ∈ D_i do
14         SD[i][j] = VarValUB[i][j]/total;
15 return SD;
```

Algorithm 4: Solution Densities

Algorithm 4 shows the pseudo code for computing $UB^{BM}(A_{x_i=j})$ for all $i = 1, \ldots, n$ and $j \in D_i$. Initially, it computes the bound for matrix A (line 1); then, for a given i, it computes γ_i and the upper bound is modified accordingly (line 3). Afterwards, for each $j \in D_i$, θ-consistency is enforced (line 7) and it iterates over the set of modified variables

(line 9-10) to compute all the γ_k that are different from 1. We store the upper bound for variable i and value j in the structure $VarValUB[i][j]$. Before computing the bound for the other variables-values the assignment $x_i = j$ needs to be undone (line 12). Finally, we normalize the upper bounds in order to correctly return solution densities (line 13-14). The time complexity is $O(mP + m\tilde{I})$.

If the matrix A is dense we expect $|\tilde{I}| \simeq n$, therefore most of the γ_k are different from 1 and need to be computed. As soon as the matrix becomes sparse enough then $|\tilde{I}| \ll n$ and only a small fraction of γ_k needs to be computed, and that is where Algorithm 4 has an edge. In pilot tests conducted over the benchmark problems presented in the experimental section, Algorithm 4 with arc consistency performed on average 25% better than the trivial implementation.

3.1.4 Counting Accuracy Analysis

We compared the algorithm based on upper bounds with the previous approaches: Rasmussen's algorithm, Furer's algorithm and the sampling algorithm we proposed in Section 3.1.2 (Algorithm 3). We generated `alldifferent` instances of size n ranging from 10 to 20 variables; variable domains were partially shrunk with a percentage of removal p varying from 20% to 80% with steps of 10%. We computed the number of solutions exactly and removed those instances that were infeasible or for which enumeration took more than 2 days (the remaining number of instances is about one thousand). As a reference, the average solution count for the `alldifferent` instances with 20% to 60% of value removals is close to one billion solutions (and up to 10 billions), with 70% of removals it decreases to few millions of solutions and with 80% of removals to few thousands.

Randomized algorithms have been run 10 times and the average of the results has been computed. In order to verify the performances with varying sampling time, we set a timeout of respectively 1, 0.1, 0.01, 0.001 seconds. On the other hand, the running time of the counting algorithm based on upper bounds is bounded by the completion of Algorithm 4.

The measures employed in the analysis are the following:

- counting error: relative error on solution count of the constraint (computed as the absolute difference between the exact solution count and the estimated one and then divided by the exact solution count)
- maximum solution density error: maximum absolute error on the solution densities (computed as the maximum of the absolute differences between the exact solution densities and the approximated ones)
- average solution density error: average absolute error on the solution densities (computed as the average of the absolute differences between the exact solution densities

and the approximated ones)

Note that we computed absolute errors for the solution densities because counting-based heuristics (see Chapter 5) usually compare the absolute value of the solution densities.

Plot 3.1 shows the counting error for the sampling algorithm, Rasmussen's and Furer's with varying timeout. Different shades of gray indicate different percentage of removals; series represent different algorithms and they are grouped based on the varying timeouts.

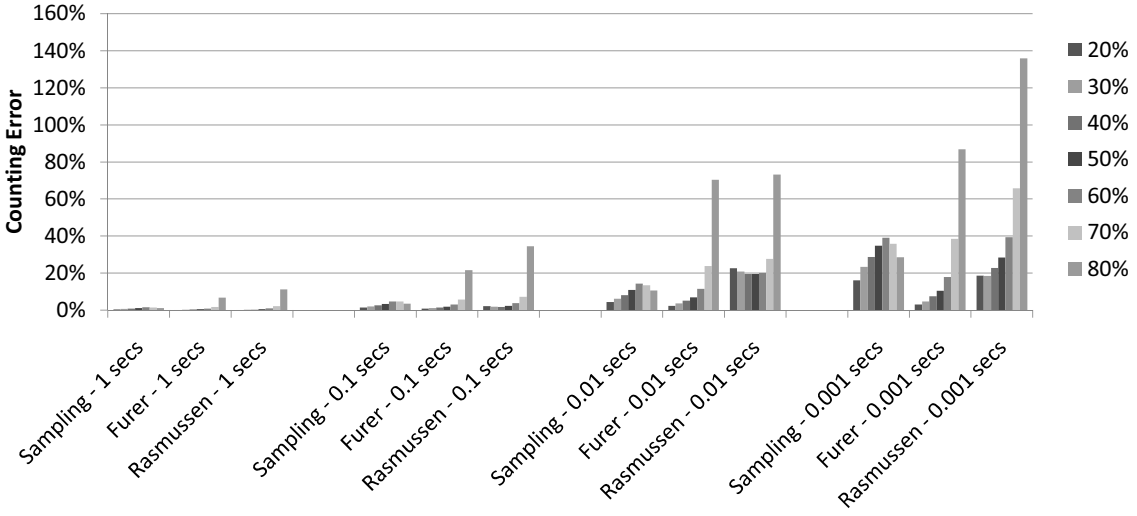


Figure 3.1 Counting Error for one thousand `alldifferent` instances with varying variable domain sizes.

The relative counting error is maintained reasonably low for 1 and 0.1 seconds of sampling, however it increases considerably if we further decrease the timeout. Note that at 0.001 the sampling algorithm reaches its limits being able to sample only few dozens of solutions (both Rasmussen's and Furer's are in the order of the hundreds of samples). We left out the results for the algorithm based on upper bounds for scaling reason: counting error varies from about 40% up to 2300% when enforcing domain consistency in Algorithm 4 (UB-DC) or 3600% with arc consistency (UB-AC) or 4800% with forward checking (UB-FC). Although being tight upper bounds, they are obviously not suitable in approximating the solution count; however it is remarkable their running times: UB-DC takes about one millisecond whereas UB-AC and UB-FC about a tenth of a millisecond (with UB-FC being slightly faster).

Despite the poor performance in approximating the solution count, they provide a very good tradeoff in approximation accuracy and computation time when deriving solution densities.

Figure 3.2 and 3.3 show respectively the maximum and average solution density errors (note that the maximum value in the y-axis is different in the two plots). Again the sampling algorithm shows a better accuracy w.r.t. Rasmussen's and Furer's. Solution density errors

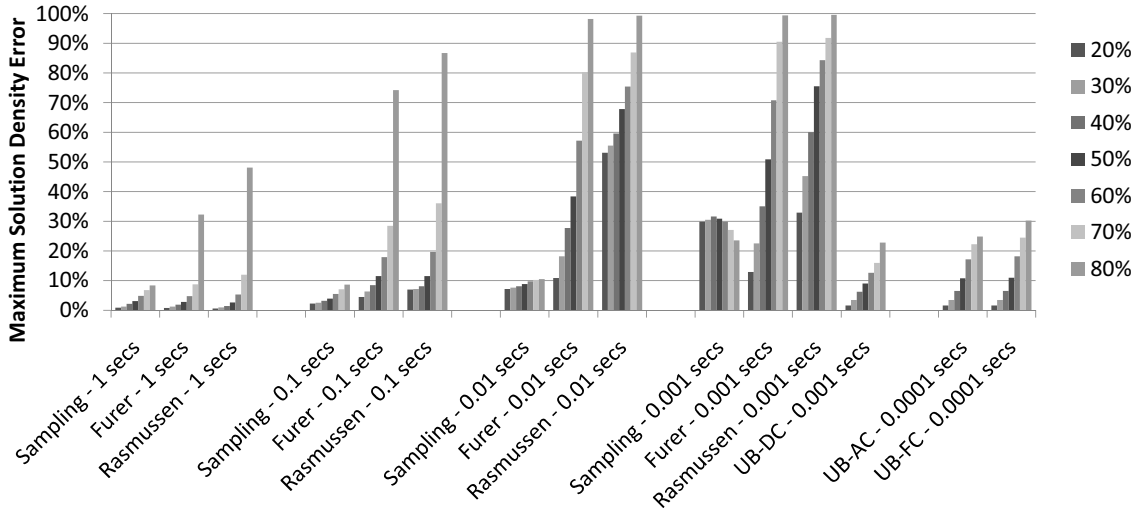


Figure 3.2 Maximum Solution Density Error for one thousand `alldifferent` instances with varying variable domain sizes.

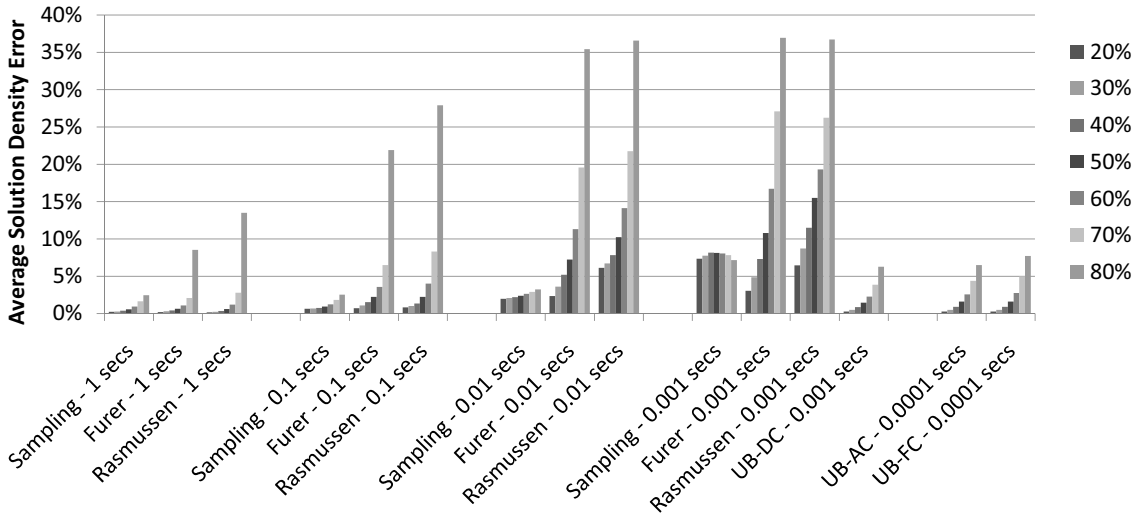


Figure 3.3 Average Solution Density Error for one thousand `alldifferent` instances with varying variable domain sizes.

are very well contained when using the upper bound approach: they are the best one when compared to the algorithms with an equivalent timeout and on average comparable to the results obtained by the sampling algorithm with a timeout of 0.01 seconds. Therefore, upper bounds offer a good accuracy despite employing just a tenth (UB-DC) or a hundredth (UB-AC, UB-FC) of the time of the sampling algorithm with comparable accuracy. Furthermore, errors for the upper bound algorithm are quite low when the domains are dense (low removal percentage) and on par with the sampling algorithm with a timeout of 0.1 or even 1 second.

Note that in the context of search heuristics dense domains are more likely to happen closer to the root of the search tree hence when it is important to have a good heuristic guidance. Finally, as expected, enforcing higher level of consistency during the local probes brings better results, however the difference between UB-DC, UB-AC and UB-FC is not striking.

3.1.5 Extending permanent upper bounds to the symmetric alldifferent constraint

Régin proposed in [87] the `symmetric_alldifferent` constraint that is a special case of the `alldifferent` in which variables and values are defined from the same set. This is equivalent to a traditional `alldifferent` with an additional set of constraints stating that variable i is assigned to a value j iff variable j is assigned to value i. This constraint is useful in many real world problems in which a set of entities needs to be paired up; particularly, in sport scheduling problems teams need to form a set of pairs that define the matches. Formally, the constraint is defined as (from [106]):

Definition 20. *Given a set of variables* $X = \{x_1, \ldots, x_n\}$ *with respective domains* $D_1, \ldots, D_n$*, the definition of the* `symmetric_alldifferent` *is:*

$$x_i = j \iff x_j = i \qquad 1 \leq i \leq n, 1 \leq j \leq n, i \neq j$$
$$\texttt{alldifferent}(X)$$

A `symmetric_alldifferent` achieving domain consistency provides more pruning power than the equivalent decomposition given by the `alldifferent` constraint and the set of $x_i = j \iff x_j = i$ constraints (see [87]). The filtering algorithm of the `symmetric_alldifferent` is inspired from the one for `alldifferent` with the difference being that the matching is computed in a graph (not necessarily bipartite) called *contracted value graph* where vertices and values representing the same entity are collapsed into a single vertex (i.e. the vertext x_i and the vertex i are merged into a single vertex i representing both the variable and the value). Régin proved that there is a bijection between a matching in the contracted value graph and a solution of the `symmetric_alldifferent` constraint. Therefore, counting the number of matchings on the contracted value graph corresponds to counting the number of solutions of the constraint.

Friedland et al. in [31] and in [2] extended the Brégman-Minc upper bound to consider the number of matchings in general undirected graphs. Therefore, we can exploit the bound as in Section 3.1.3 in order to provide an upper bound of the solution count and the solution densities for the `symmetric_alldifferent` constraint. The upper bound for the number of matchings of a graph $G = (V, E)$, representing the contracted value graph, is the following:

$$\#matchings(G) \leq \prod_{v \in V} (deg(v))!^{\frac{1}{2deg(v)}} \tag{3.7}$$

where $deg(v)$ is the degree of the vertex v and $\#matchings(G)$ denotes the number of matchings on the graph G. Note that in the case of a bipartite graph, this bound is equivalent to the Brégman-Minc upper bound.

The algorithm for counting the number of solutions and computing the solution densities can be easily derived from what we proposed for the `alldifferent`.

Example 7. *Consider a* `symmetric_alldifferent` *defined on six variables* $x_1, \ldots, x_6$ *each one having a domain equal to* $\{1, \ldots, 6\}$*. In this case, the number of solutions of the* `symmetric_alldifferent` *can be computed as* $5 * 3 = 15$*. In the contracted value graph each vertex is connected to each other vertex, forming a clique of size 6, therefore all the vertices have a degree equal to 5. The upper bound proposed by Friedland is equal to:*

$$\#matchings(G) \leq \prod_{v \in V} (deg(v))!^{\frac{1}{2deg(v)}} = (5!^{1/10})^6 \approx 17.68$$

In the `alldifferent` *formulation, the related value graph has variable vertices connected to each of the values (from 1 to 6) thus the* r_i*'s are equal to 6. If we consider to rule out all the edges causing degenerated assignments* ($x_i = i$) *then we end up with a value graph in which all the* r_i*'s are equal to 5. The Brégman-Minc upper bound would give:*

$$perm(A) \leq \prod_{i=1}^{n} (r_i!)^{1/r_i} = (5!^{(1/5)})^6 \approx 312.62.$$

The result is obviously very far from the upper bound given by Formula 3.7 as well as from the exact value.

3.2 Counting for Global Cardinality Constraint

We present in this section how to extend the results obtained in Section 3.1.3 to the Global Cardinality Constraint (GCC) that, we recall, is a generalization of the `alldifferent` constraint.

Definition 21 (Global Cardinality Constraint)**.** *The set of feasible tuples of a constraint* $\texttt{gcc}(X, l, u)$ *where* X *is a set of* n *variables,* l *and* u *respectively the lower and upper bounds for each value, is defined as:*

$$T(gcc(X, l, u)) = \{(d_1, \ldots, d_n) \mid d_i \in D_i, l_d \leq |\{d_i | d_i = d\}| \leq u_d \, \forall d \in D_X\}$$

We will consider a GCC in which all the bounded variables are removed and the lower and upper bounds are adjusted accordingly (the semantics of the constraint are unchanged). We refer to the new set of variables as $X' = \{x \in X \mid x \text{ is not bound}\}$; lower bounds are l' where $l'_d = \max(l_d - |\{x \in X \mid x = d\}|, 0)$ and upper bounds u' are defined as $u'_d = u_d - |\{x \in X \mid x = d\}|$; we assume the constraint is feasible therefore $u'_d \geq 0$ for each $d \in D_X$ (i.e. the upper bound on the value occurrences is respected).

Inspired by [83] and [111], we define G_l the lower bound graph.

Definition 22. *Let $G_l(X' \cup D_l, E_l)$ be an undirected bipartite graph such that one partition represents the unbounded variable set and the other the extended value set, that is for each $d \in D_X$ the graph has l'_d vertices representing d (l'_d possibly equal to zero). There is an edge $(x_i, d) \in E_l$ if and only if $d \in D_i$; in case the vertex representing d is duplicated (i.e. $l'_d > 1$) then x_i is connected to each copy.*

Note that a maximum matching on G_l corresponds to a partial assignment of the variables in X that satisfies the GCC lower bound constraint. This partial assignment may or may not be completed to a full assignment that satisfies both upper bound and lower bound occurrence constraints (here we do not take into consideration augmenting paths as in [111] but instead we fix the variables to the values represented by the matching in G_l).

Example 8. *Suppose we have a GCC defined on $X = \{x_1, \ldots, x_6\}$ with domains $D_1 = D_4 = \{1, 2, 3\}$, $D_2 = \{2\}$, $D_3 = D_5 = \{1, 2\}$ and $D_6 = \{1, 3\}$; lower and upper bounds for the values are respectively $l_1 = 1$, $l_2 = 3$, $l_3 = 0$ and $u_1 = 2$, $u_2 = 3$, $u_3 = 2$. Considering that $x_2 = 2$, the lower and upper bounds for the value 2 are respectively $l'_2 = 2$ and $u'_2 = 2$. The lower bound graph is shown in Figure 3.4a: variable x_2 is bounded and thus does not appear in the graph, value vertex 2 is represented by two vertices because it has $l'_2 = 2$ (although $l_2 = 3$); finally value vertex 3 does not appear because it has a lower bound equal to zero. The matching shown in the figure (bold edges) is maximum however if we fix the assignments represented by it ($x_1 = 2$, $x_4 = 2$, $x_6 = 1$) it is not possible to have a consistent solution since both x_3 and x_5 have to be assigned either to 1 or 2 hence exceeding the upper bound constraint. To compute the permanent two additional fake value vertices would be added to the graph and connected to all the variable vertices (not shown in the figure), as explained in Theorem 14.*

Every partial assignment that satisfies just the lower bound constraint might correspond to several maximum matchings in G_l due to the duplicated vertices. For each partial assignment satisfying the lower bound constraint there are exactly $\prod_{d \in D_X} l'_d!$ maximum matchings corresponding to that particular partial assignment. If we take into consideration Example 8

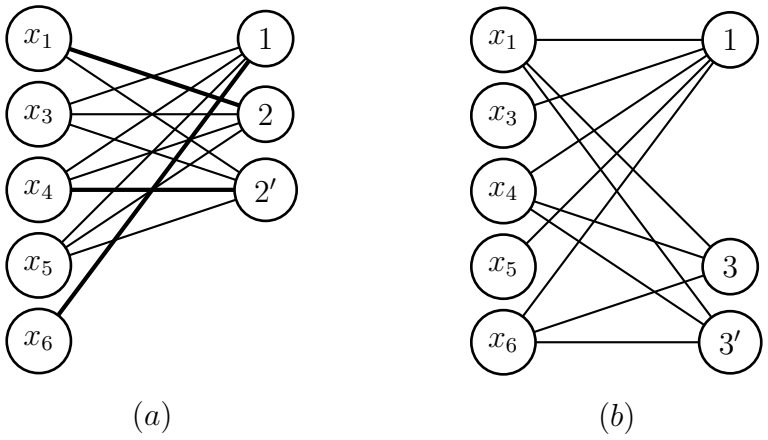


Figure 3.4 Lower Bound Graph (a) and Residual Upper Bound Graph (b) for Example 8

shown in Figure 3.4a, variables x_1 and x_4 may be matched respectively to any permutation of the vertices 2 and 2′, however no matter which is the permutation, this set of matchings represents always the assignment of both x_2 and x_4 to the value 2.

Let M_l [7] be the set of maximum matchings in G_l. We define $f : M_l \rightarrow \mathbb{N}$, a function that counts the number of possible ways a maximum matching can be extended to a full GCC solution. As shown in Example 1, f can be possibly equal to zero. Note that the number of the remaining variables that need to be assigned starting from a matching $m \in M_l$ is equal to $K = |X'| - \sum_{d \in D_X} l'_d$.

The total number of solutions satisfying the GCC constraint is:

$$\#gcc(X, l, u) = \frac{\sum_{m \in M_l} f(m)}{\prod_{d \in D_X} l'_d!} \leq \frac{|M_l| \max_{m \in M_l}(f(m))}{\prod_{d \in D_X} l'_d!} \leq \frac{UB(G_l) \max_{m \in M_l}(f(m))}{\prod_{d \in D_X} l'_d!} \quad (3.8)$$

Computing $f(m)$ turns out to be equivalent to the task of computing the number of matchings in a graph (where the matched variables and values are removed) therefore in the most general case it is a #P-complete problem on its own; we focus then on upper bounding $f(m)$.

In order to do that, we introduce the upper bound residual graph. Intuitively, it is similar to the lower bound graph but it considers the upper bound constraint.

Definition 23. *Let $G_u(X' \cup D_u, E_u)$ be an undirected bipartite graph such that one partition represents the unbounded variable set and the other the extended value set, that is for each $d \in D_X$ the graph has $u'_d - l'_d$ vertices representing d (if $u'_d - l'_d$ is equal to zero then there is no vertex representing d). There is an edge $(x_i, d) \in E_u$ if and only if $d \in D_i$ and $u'_d - l'_d > 0$;*

7. if $\forall d \in D_X, l'_d = 0$ then $M_l = \{\emptyset\}$ and $|M_l| = 1$

in case the vertex representing d is duplicated then x_i is connected to each copy.

Similarly to the lower bound matching, a matching on G_u that covers K variables may or may not be completed to a full assignment satisfying the complete GCC. Figure 3.4b shows the residual upper bound graph for Example 1: value 2 disappears from the graph since it has $u'_2 = l'_2$ i.e. starting from a matching in the lower bound graph, the constraints on value 2 are already satisfied.

In order to compute $\max_{m \in M_l}(f(m))$, we should build $\binom{|X|}{K}$ graphs each with a combination of K variables, and then choose the one that maximizes the permanent. More practically, given the nature of the UB^{MB} and UB^{LB}, it suffices to choose K variables which contribute with the highest factor in the computation of the upper bounds; this can be easily done in $O(n \log K)$ by iterating over the n variables and maintining a heap with K entries with the highest factor. We write $\hat{G}_u$ and $\tilde{G}_u$ for the graphs in which only the K variables that maximize respectively UB^{MB} and UB^{LB} are present; note that $\hat{G}_u$ might be different from $\tilde{G}_u$.

We recall here that due to Theorem 1 although only K variables are chosen, the graphs $\hat{G}_u$ and $\tilde{G}_u$ are completed with fake vertices in such a way to have an equal number of vertices on the two vertex partitions. Thus, as in the lower bound graph, the given upper bound has to be scaled down by a factor of $\prod_{d \in D_X}(u'_d - l'_d)!$. From Equation 3.8, the number of GCC solutions is bounded from above by:

$$\#gcc(X, l, u) \leq \frac{UB(G_l) \min(UB^{MB}(\hat{G}_u), UB^{LB}(\tilde{G}_u))}{\prod_{d \in D_X}(l'_d!(u'_d - l'_d)!)} \tag{3.9}$$

Scaling and also fake vertices used with the permanent bounds are factors that degrade the quality of the upper bound. Nonetheless, solution densities are computed as a ratio between two upper bounds therefore these scaling factors are often attenuated.

Example 9. *We refer to the GCC constraint described in Example 8. The exact number of solutions is 19. The UB^{MB} and UB^{LB} for the lower bound graph in Figure 3.4a are both* 35 *(the scaling for the two fake value vertices is already considered). In the upper bound only 2 variables need to be assigned and the one maximizing the bounds are x_1 and x_4 (or possibly x_6): the resulting permanent upper bound is* 6*. An upper bound on the total number of GCC solutions is then $\lfloor \frac{35*6}{4} \rfloor = 52$ where the division by 4 is due to $l'_2! = 2!$ and $u'_3! = 2!$.*

Figure 3.5 shows the lower bound and residual upper bound graph for the same constraint where $x_1 = 1$ and domain consistency is achieved. Vertex x_1 has been removed and $l'_1 = 0$ and $u'_1 = 1$. The graph G_l has a permanent upper bound of 6. The number of unassigned variables in G_u is 2 and the ones maximizing the upper bounds are x_4 and x_6, giving an upper bound of 6. The total number of GCC solutions with $x_1 = 1$ is then bounded above

by $\lfloor \frac{6*6}{4} \rfloor = 9$*; the approximate solution density before normalizing it is thus* $9/52$*. Note that after normalization, it turns out to be about* 0.18 *whereas the exact computation of it is* $5/19 \sim 0.26$*.*

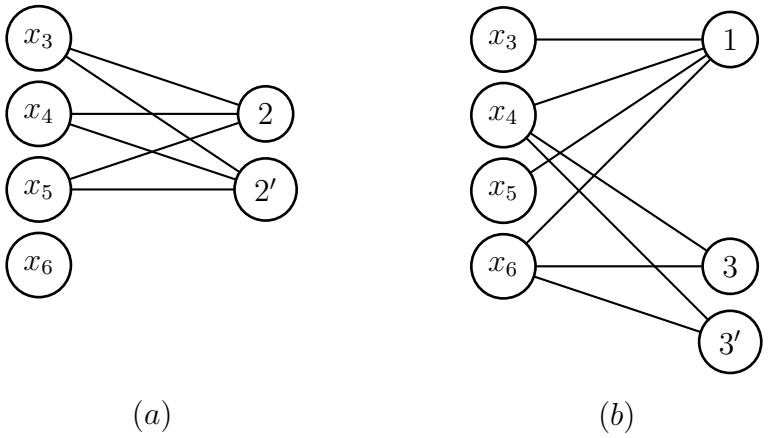


Figure 3.5 Lower Bound Graph (a) and Residual Upper Bound Graph (b) assuming $x_1 = 1$

3.3 Counting for Regular Constraints

The `regular`(X, Π) constraint [77] holds if the values taken by the sequence of finite domain variables $X = \langle x_1, x_2, \ldots, x_n \rangle$ spell out a word belonging to the regular language defined by the deterministic finite automaton $\Pi = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is a partial transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states. The filtering algorithm associated to this constraint is based on the computation of paths in a graph. The automaton is unfolded into a layered acyclic directed graph $G = (V, A)$ where vertices of a layer correspond to states of the automaton and arcs represent variable-value pairs. We denote by $v_{\ell,q}$ the vertex corresponding to state q in layer ℓ. The first layer only contains one vertex, v_{1,q_0}; the last layer only contains vertices corresponding to accepting states, $v_{n+1,q}$ with $q \in F$. This graph has the property that paths from the first layer to the last are in one-to-one correspondence with solutions of the constraint. The existence of a path through a given arc thus constitutes a support for the corresponding variable-value pair [77]. Figure 3.6 gives an example of a layered directed graph built for one such constraint on five variables; layers are vertical and denoted by $L_1, \ldots, L_6$, vertices within a layer correspond to states of the automaton. An arc joining a vertex of layer L_i to another of layer L_{i+1} represents a feasible value for variable x_i: the arc's colour stands for the value.

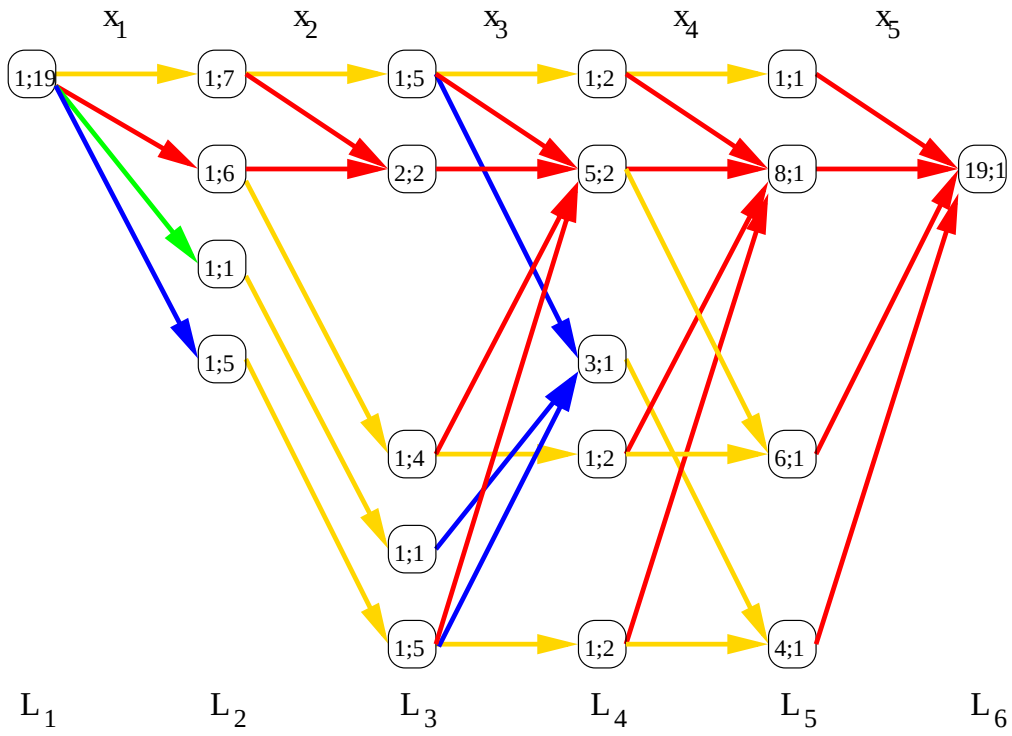


Figure 3.6 The layered directed graph built for a `regular` constraint on five variables. Vertex labels represent the number of incoming and outgoing paths.

The time complexity of the filtering algorithm is linear in the size of the graph (the number of variables times the number of transitions appearing in the automaton). Essentially, one forward and one backward sweep of the graph are sufficient. An incremental version of the algorithm, which updates the graph as the computation proceeds, has a time complexity that is linear in the size of the changes to the graph (see [77]).

3.3.1 Counting Paths in the Associated Graph

Given the graph built by the filtering algorithm for `regular`, what is the additional computational cost of determining its number of solutions? As we already pointed out, every (complete) path in that graph corresponds to a solution. Therefore it is sufficient to count the number of such paths. We express this through a simple recurrence relation, which we can compute by dynamic programming. Let $\#op(\ell, q)$ denote the number of paths from $v_{\ell,q}$

to a vertex in the last layer. Then we have:

$$\begin{aligned} \#op(n+1,q) &= 1 \\ \#op(\ell,q) &= \sum_{(v_{\ell,q},v_{\ell+1,q'})\in A} \#op(\ell+1,q'), \quad 1 \le \ell \le n \end{aligned}$$

The total number of paths is given by

$$\#\texttt{regular}(X,\Pi) = \#op(1,q_0)$$

in time linear in the size of the graph even though there may be exponentially many of them. Therefore this is absorbed in the asymptotic complexity of the filtering algorithm.

The search heuristics we consider require not only *solution counts* of constraints but *solution densities* of variable-value pairs as well. In the graph of `regular`, such a pair (x_i, d) is represented by the arcs between layers i and $i+1$ corresponding to transitions on value d. The number of solutions in which $x_i = d$ is thus equal to the number of paths going through one of those arcs. Consider one such arc $(v_{i,q}, v_{i+1,q'})$: the number of paths through it is the product of the number of outgoing paths from $v_{i+1,q'}$ and the number of incoming paths to $v_{i,q}$. The former is $\#op(i+1,q')$ and the latter, $\#ip(i,q)$, is just as easily computed:

$$\begin{aligned} \#ip(1,q_0) &= 1 \\ \#ip(\ell+1,q') &= \sum_{(v_{\ell,q},v_{\ell+1,q'})\in A} \#ip(\ell,q), \quad 1 \le \ell \le n \end{aligned}$$

where $\#ip(\ell,q)$ denotes the number of paths from v_{1,q_0} to $v_{\ell,q}$.

In Figure 3.6, the left and right labels inside each vertex give the number of incoming and outgoing paths for that vertex, respectively. For example, the arc between the vertex labeled "$2;2$" in layer L_3 and the vertex labeled "$5;2$" in layer L_4 has $2 \times 2 = 4$ paths through it.

Let $A(i,d) \subset A$ denote the set of arcs representing variable-value pair (x_i, d). The solution density of pair (x_i, d) is thus given by:

$$\sigma(x_i, d, \texttt{regular}) = \frac{\sum_{(v_{i,q},v_{i+1,q'})\in A(i,d)} \#ip(i,q) \cdot \#op(i+1,q')}{\#op(1,q_0)}$$

Once these quantities are tabulated, the cost of computing the solution density of a given pair is in the worst case linear in $|Q|$, the number of states of the automaton.

3.3.2 An Incremental Version

Because a constraint's filtering algorithm is called on frequently, the graph for `regular` is not created from scratch every time but updated at every call. Given that we already maintain data structures to perform incremental filtering for `regular`, should we do the same when determining its solution count and solution densities?

For the purposes of the filtering algorithm, as one or several arcs are removed between two given layers of the graph as a consequence of a value being deleted from the domain of a variable, other arcs are considered for removal in the previous (respectively following) layers only if the out-degree (respectively in-degree) of some vertices at the endpoints of the removed arcs becomes null. Otherwise no further updates need to be propagated. Consequently even though the total amount of work in the worst case is bounded from above by the size of the graph, it is often much less in practice.

In the case of solution counting, the labels that we added at vertices contain finer-grained information requiring more extensive updates. Removing an arc will change the labels of its endpoints but also those of every vertex reachable downstream and of every vertex upstream which can reach that arc. Here the total amount of work in practice may be closer to the worst case (linear on the size of the graph). Therefore maintaining the additional data structures could prove to be too expensive.

3.3.3 A Lazy Evaluation Version

Rather than keeping updated information on $\#op()$ and $\#ip()$ throughout the solving process (that is during both constraint propagation and branching), it is possible to compute them on an as-needed basis i.e. just before branching. The advantage is twofold: firstly we avoid computing counting information systematically for each constraint, which might end up being unnecessary; secondly we avoid updating several times the counting information (during propagation) before it is actually used (during branching). Counting is thus performed only when a request for the solution count is received. Furthermore counting information is recomputed and cached if and only if a domain event has occurred in the constraint. The request for the solution count triggers the computation of the required $\#op()$ and $\#ip()$ values. If no change in the constraint occurred since those values were last computed, they are simply looked up in a table. Otherwise they are computed iteratively and cached before they are returned to avoid recomputing them.

On some pilot tests, the lazy evaluation version was faster than the version computing from scratch and up to five times faster than the version maintaining the data structures.

3.4 Summary

In this chapter, we propose counting algorithms for some families of global constraints. Such counting algorithm will be exploited in the following chapter for designing some efficient search heuristics. Particularly, in Section 3.1 we proposed two counting algorithms for the `alldifferent` constraint. The first one based on sampling provides more accurate information but it is more time consuming. The second one is a good trade-off in time and accuracy. Section 3.2 extends this latter result to the `gcc` constraint. Finally, a counting algorithm for `regular` constraint has been proposed in 3.3.

CHAPTER 4

Solution Counting Based Heuristics

Despite many research efforts to design generic and robust search heuristics and to analyze their behaviour, a successful CP application often requires customized, *problem-centered* search heuristics or at the very least some fine tuning of standard ones, particularly for value selection [1]. In contrast, Mixed Integer Programming (MIP) and SAT solvers feature successful default search heuristics that basically reduce the problem at hand to a modeling issue.

Constraints have played a central role in CP because they capture key substructures of a problem and efficiently exploit them to boost inference. In this chapter we present work that intends to do the same thing for search, proposing *constraint-centered counting-based* heuristics. A constraint's consistency algorithm often maintains data structures in order to incrementally filter out values that are not supported by the constraint's set of valid tuples. These same data structures may be exploited to evaluate *how many* valid tuples there are (see Chapter 3). Up to now, the only visible effect of the consistency algorithms has been on the domains, projecting the set of tuples on each of the variables. Additional information about the number of solutions of a constraint can help a search heuristic to focus on critical parts of a problem or promising solution fragments. Polynomial time approximate or exact algorithms to count the number of solutions of several common families of constraints were given in Chapter 3, [78], [113] and [79].

Evaluating the number of solutions of Boolean formulas and of CSPs has received considerable attention lately ([84], [94], [39]). Gomes et al. [43] and [42] compute bounds on the number of solutions of SAT instances (later on extended to CSP [47]) by adding to the problem so-called streamlining constraints. Gogate and Dechter [40] propose to sample solutions of a problem in order to estimate the number of solutions; they then use a Sampling/Importance Resampling technique in [41] to obtain uniform sampling approximation guarantees. Such works share our interest in evaluating how many solutions involve a particular variable assignment, however the main difference between our work and these is that we focus on individual constraints whereas they consider the problem as a whole. Counting on individual constraints gives more precise information (exact or approximated) but it lacks the global view of the problem; nevertheless the work cited above requires finding solutions of the problem making them useless to design search heuristics (since a solution has already

1. portions of this chapter appeared in [113], [114] and [116]

been found).

4.1 Background

There is a large body of scientific literature on search heuristics to solve CSPs. Heuristics are usually classified in two main categories: static variable ordering heuristics (SVOs) and dynamic variable ordering heuristics (DVOs). The former decided an ordering at the beginning of the search tree that is kept fixed during the search. Common SVOs order the variables according to lexicographic order lexico, decreasing degree (i.e. number of constraints in which a variable is involved) deg.

DVOs are commonly considered more effective as they exploit information gathered during search. They often obey the *fail-first principle* originally introduced by Haralick and Elliott in [51] i.e. "To succeed, try first where you are most likely to fail". The same authors proposed the widely-used heuristic mindom (referred to also as dom) in which the variable with the smallest remaining domain is choosen for branching; the aim of such heuristic is to minimize the branch depth. A similar heuristic, proposed by Brélaz in [17], selects the variable with the smallest remaining domain and then breaks ties by choosing the one with the highest *dynamic degree* - ddeg [2] (that is the one constraining the largest number of unbound variables). Bessière et al. in [13] and Smith et al. in [97] combined the domain and degree information that is choosing the variable that minimizes the ratio dom/deg or dom/ddeg.
In [97], Smith et al. tried to push even further the fail-first principle and proposed four different heuristics that consider also the tightness of the constraints adjacent to the variable to branch on. These heuristics were reviewed and thoroughly tested in [8] where the authors shed some light on the relationship between heuristics and adherence to the fail-first policy (as minimization of branch depth or size of infeasible subtrees).
Gent et al. formally introduced in [33] the concept of constrainedness κ:

$$\kappa = \frac{-\sum_{c \in C} \log_2(1 - p_c)}{\sum_{x_i \in X} \log_2(|D_{x_i}|)}$$

where p_c is the proportion of value combinations ruled out by constraint c. Constrainedness characterizes the phase transition of random constraint networks and it can be successfully exploited to design search heuristics. In [33] and [34], the authors designed three heuristics derived from κ that either: maximize the problem solution density, maximize the expected number of solutions, or minimize κ (i.e. branching towards the most unconstrained subproblem). These works resemble in their essence what we are proposing here, with the

2. Also referred to as *future degree* or *forward degree* in the literature.

main difference being that we consider global constraints whereas they implicitly consider constraints with low arity or constraints described in extension.

In [10], Bessière et al. proposed a multilevel variable ordering in which heuristic information about variable neighborhoods is also taken into account when selecting a variable. This approach is general and it can be built on top of any variable ordering heuristic. Formally:

$$\begin{aligned} H^{\circledcirc}_{(0,\alpha)}(x_i) &= \alpha(x_i) \\ H^{\circledcirc}_{(k,\alpha)}(x_i) &= \frac{\sum_{x_j \in \Gamma(x_i)} (\alpha(x_i) \circledcirc H^{\circledcirc}_{(k-1,\alpha)}(x_j)}{|\Gamma(x_i)|^2} \end{aligned}$$

where α can be any heuristic like dom, deg or dom/deg; $\Gamma(x_i)$ is the set of variables that share a constraint with x_i and $\circledcirc$ is an operator such as $\{+, \times\}$.

Impact-based heuristics. Refalo proposed in [86] Impact Based Search (IBS), a heuristic that chooses the variable whose instantiation triggers the largest search space reduction (highest impact) that is approximated as the reduction of the product of the variable domain cardinalities. More formally the impact of a variable-value pair is:

$$I(x_i = a) = 1 - \frac{P_{after}}{P_{before}}$$

where P_{after} and P_{before} are the products of the domain cardinalities respectively after and before the branching $x_i = a$. The impact is either computed exactly at a given node of the search (the exact computation provides better information but is more time consuming) or approximated as the average reduction observed during the search (hence automatically collected on-the-go at almost no additional cost), that is:

$$\bar{I}(x_i = a) = \frac{\sum_{k \in K} I^k(x_i = a)}{|K|}$$

where K is the index set of the impact observed so far for the assignment $x_i = a$. The variable impact [3] is:

$$\mathcal{I}(x_i) = 1 - \sum_{a \in D'_{x_i}} (1 - \bar{I}(x_i = a))$$

where D'_{x_i} is the current domain of the variable x_i. Impact initialization is fundamental to obtain good performances starting from the root of the search tree; therefore, Refalo proposed to initialize the impacts by probing each variable-value pair at the root node (note that this subsumes a reduced form of singleton consistency at the root node). In case domains are

3. this slightly differs from [86], as we follow [22]

large, the initialization might become a heavy task; to overcome this limitation, it is possible to partition the domain of a variable in k subsets ($D_{x_i} = D^1_{x_i} \cup \cdots \cup D^k_{x_i}$) and to probe each subset instead of each variable-value pair; the impact value is then shared among the values belonging to that subset.

IBS selects the variable having the largest impact (hence trying to maximize the propagation effects and the reduction of the search space) and then selects the value having the smallest impact (hence leaving more choices for the future variables); *node impacts* (i.e. exact impacts computed at a given node) may be used as a tie breaker on a subset of variables. When IBS is employed along with randomized restart techniques, impacts can be profitably reused over different restarts as they benefit from what was learned in previous runs. Correia and Barahona in [22] extended IBS by systematically enforcing a reduced form of singleton consistency (see Section 5). They called this heuristic Reduced Singleton Consistency with Look Ahead (RSC+LA). At a given node, impacts are computed exactly and used in a similar manner as in [86], however if a probe for a variable-value pair causes a domain wipe-out then it is filtered out. To reduce the overhead caused by maintaining restricted singleton (generalized) arc consistency, the authors proposed a second heuristic in which node impacts are computed only for those variables that have minimum domain cardinalities (RSC2+LA).

As an interesting connection with impact-based heuristics, in [102] Szymanek and O'Sullivan proposed to query the model constraints to approximate the number of filtered values by each constraint individually; this can be viewed as a form of constraint-level impact. This information is then exploited to design a variable and/or value selection heuristic. Unfortunately, their work is limited only to the `permutation` constraint (a special case of the `alldifferent` constraint) and the `sum` constraint.

Conflict-driven heuristics. In [14], Boussemart et al. proposed a conflict-driven variable ordering heuristic: they extended the concept of *variable degree* integrating a simple but effective learning technique that takes into account failures. Basically, each constraint has a weight associated that is increased by one each time the constraint leads to a failure (i.e. a domain wipe-out). A variable has a *weighted degree* – wdeg – that is the sum of the constraint weights in which it is involved. Formally, the weighted degree of a variable is:

$$\alpha_{wdeg}(x_i) = \sum_{C \in \mathcal{C}} weight[C] \quad | \quad Vars(C) \ni x_i \wedge |FutVars(C)| > 1$$

where $FutVars(C)$ denotes the uninstantiated variables of the constraint C, $weight[C]$ is its weight and $Vars(C)$ the variables involved in C. The heuristics proposed simply choose the variable that maximize wdeg or the one that minimize $dom/wdeg$.

Grimes and Wallace in [48] and later in [49] proposed a few improvements over dom/wdeg. Firstly in [48] they noticed that weights carried along over different restarts are not particularly informative; in fact variables involved in constraints that caused failures in a first run are likely to be chosen first in a second run, therefore they probably will not have the weights of their constraints increased again. On the other hand, they noticed that several random probes allow to fruitfully initialize constraint weights; the key point here is that the complete randomness in the variable and value selection does not introduce any bias as opposed to randomized restarts where dom/wdeg is employed in each restart.
A second interesting contribution of [49] has been to detect and to accordingly increase the weights of constraints causing not only domain wipe-outs but also value deletions; experimental results did not show a real breakthrough w.r.t. dom/wdeg, nonetheless this approach inspired some follow-up works such as [4] described later.

We mention here a few drawbacks related to dom/wdeg heuristics: there is no general method to deal with global constraints [4], and the heuristic is particularly sensitive to revision orderings (i.e. the ordering of the propagation queue) hence leading to varying performance. The latter aspect may affect constraint weights in several ways, for example:

- the integrated effect of value deletions of two (or more) constraints can cause a failure but independently none of them can detect the inconsistency. However, depending on the revision ordering only one of the two will have an increase of its weight.
- two (or more) constraints can cause independently a domain wipe-out, however the search backtracks as soon as one of them occurs. Once again, the result is that only one of the constraints will increase its weight.

Constraint weights need to be maintained throughout the search tree in order to reflect the learning-from-failure performed during search. However, different parts of the search tree may have different criticalities hence a constraint that is crucial (i.e. with a high weight) in a branch may be trivially satisfiable in another one.
The concerns briefly described above have been tackled in [4] by Balafoutis and Stergiou and they proposed three improvements over the original dom/wdeg. Firstly, they consider value deletions as in [49], although with some slight differences; in case of a failure they increase the weights of only those constraints that concurred in the domain wipe-out; three heuristics were proposed: increase by one all the weights of the constraints that caused at least one value deletion, or increase the weights proportionally (normalized or not) to the number of value deletions.
Secondly, they introduced a variant of dom/wdeg (called *fully assigned*) in which in case of a domain wipe-out all the constraints that caused at least a value deletion (on no matter

4. Personal communication with Christophe Lecoutre

which variable domain) get their weights increased. This technique was originally conceived to partially overcome the problem of detecting only one constraint causing a domain wipe-out.
Thirdly, inspired by some SAT solvers, they adopted *weight aging*, that is the constraint weights are periodically reduced. This limits the inertia of constraints that got a significant weight early in the search but that are not critical anymore later on.

Some problem classes benefit from the variants of dom/wdeg, although the advantage often remains well within one order of magnitude and none of them clearly outperforms the original heuristic across all the problem classes [3].
Nowadays, conflict-driven heuristics such as dom/wdeg and heuristics based on impacts are considered to be the state-of-the-art of generic heuristics [3], [23] without any of the two outperforming clearly the other.

Approximated counting-based heuristics The idea of using an approximation on the number of solutions of a problem as heuristic is not new.

Dechter et al. in [25] analyze the constraint network to identify features that render the problem solvable in a backtrack-free fashion and in polynomial time. They propose algorithms to count the number of solutions for these classes of easy problems. In complex problems (that is, problems that cannot be solved in a backtrack-free fashion), they suggest to simplify the constraint network in such a way that becomes easily countable; counting information on the whole simplified problem is then exploited to guide the value selection on the original problem.

In [73], Meisels et al. make use of Bayes network to approximate the number of solutions of a CSP. In case of tree structured CSPs the proposed approach is equivalent to the one proposed by Dechter et al. in [25] but it performs better for general constraint networks. They also suggest the use of marginals provided by the Bayes network to guide the search.

Kask et al. [61] approximate the total number of solutions extending a partial solution to a CSP and use it in a value selection heuristic, choosing the value whose assignment to the current variable gives the largest approximate solution count. An implementation optimized for binary constraints performs well compared to other popular strategies.

Hsu et al. [54] apply a Belief Propagation algorithm within an Expectation Maximization framework (EMBP) in order to approximate variable biases (or marginals) i.e. the probability a variable takes a given value in a solution. Even though the approach is general, they derived formulas for computing and updating the variable biases only for the `alldifferent`

constraint and they experimented on the Quasigroup with Holes Problem. The computation of variable biases is time-consuming but they show that the heuristic is effective even when it is employed only in the top part of the search tree. This work has been recently extended by Le Bras et al. in [15]; the contribution is twofold: firstly we generalized Hsu et al.'s method to tackle any global constraint; secondly we leveraged the same method to consider the problem model as an individual (likely NP-Hard) constraint. The second method shows promising results, however it requires to probe each variable-value pair at each node of the search tree to compute the variable biases (hence subsuming a restricted form of Singleton Consistency); in this respect it resembles the RSC+LA heuristic and it actually improves over it, nonetheless the significant probing overhead makes it suitable only for a restricted class of problems.

Note however that the main difference between our work and the one cited previously is that we focus on fine-grained information on individual constraints whereas the previous work coarse information on the whole problem.

Finally, we end this section with a remark: although heuristics have been extensively studied, in practice in real-life problems the user is likely to use custom heuristics in order to obtain significant results. Most of the time the heuristics proposed lack either in generality or in performance, therefore leaving open the quest for generic and performing heuristics.

4.2 Generic Constraint-Centered Counting-based Heuristics

Whereas most generic dynamic search heuristics in constraint programming rely on information at the fine-grained level of the individual variable (e.g. its domain size and degree), we investigate dynamic search heuristics based on coarser, but more global, information. Global constraints are successful because they encapsulate powerful specialized filtering algorithms but firstly because they bring out the underlying structure of combinatorial problems. That exposed structure can also be exploited during search. The heuristics proposed here revolve around the knowledge of the number of solutions of individual constraints, part of the intuition being that a constraint with few solutions corresponds to a critical part of the problem with respect to satisfiability. We review here some concepts presented in Chapter 3:

Definition 24 (solution count)**.** *Given a constraint $\gamma(x_1, \ldots, x_k)$ and respective finite domains $D_{i\ 1\leq i\leq k}$, let $\#\gamma(x_1, \ldots, x_k)$ denote the number of solutions of constraint γ.*

Search heuristics following the *fail-first principle* (detect failure as early as possible) and centered on constraints can be guided by a count of the number of solutions left for each constraint. We might for example focus the search on the constraint currently having the

smallest number of solutions, recognizing that failure necessarily occurs through a constraint admitting no more solution.

Such information can be misleading though, since two constraints may have different arities or their respective variables may have domains of vastly different sizes. An extreme example of this would be a binary constraint whose variables have a domain size of 2 and 4 possible solutions: this solution count is low but in fact the constraint allows every possible combination of values for its variables and so is not constraining at all. Alternately we can compute the ratio of the solution count of a constraint to the size of the Cartesian product of the appropriate domains, in a way measuring the tightness of the projection of the constraint onto the individual variables.

Definition 25 (projection tightness [78])**.** *Given a constraint* $\gamma(x_1, \ldots, x_k)$ *and respective finite domains* $D_{i\ 1\leq i\leq k}$*, we will call*

$$T_\gamma = \frac{\#\gamma(x_1, \ldots, x_k)}{\prod_{1\leq i\leq k} |D_i|}$$

the projection tightness *of constraint* γ*.*

We can go one step further with solution count information and evaluate it for each variable-value pair in an individual constraint.

Definition 26 (solution density)**.** *Given a constraint* $\gamma(x_1, \ldots, x_k)$*, respective finite domains* $D_{i\ 1\leq i\leq k}$*, a variable* x_i *in the scope of* γ*, and a value* $d \in D_i$*, we will call*

$$\sigma(x_i, d, \gamma) = \frac{\#\gamma(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_k)}{\#\gamma(x_1, \ldots, x_k)}$$

the solution density [5] *of pair* (x_i, d) *in* γ*. It measures how often a certain assignment is part of a solution.*

We can for example favour high solution densities with the hope that such a choice generally brings us closer to satisfying the whole CSP. Our choice may combine information from every constraint in the model, be restricted to a single constraint, or even to a given subset of variables.

The algorithms proposed in the following sections define the search heuristics with which we will experiment in Section 4.3. In the following, we denote by $\Gamma(x_i)$ the set of constraints whose scope contains the variable x_i and with D_γ the size of the Cartesian product of the domains of the variables in the constraint's scope i.e. $D_\gamma = \prod_{x_i \in X(\gamma)} |D_{x_i}|$ where we recall

5. Also referred to as *marginal* in some of the literature.

$X(\gamma)$ denotes the variables involved in the constraint γ. All the heuristics proposed assume a lexicographical ordering as tie breaking (although it is very unlikely that two variable-value pairs have the same solution densities). Note also that counting information is cached during search individually for each constraint and updated iff a domain change occured in a variable within the scope of the constraint considered.

We give in the following a brief overview of the heuristics proposed that will be fully described in the next sections:

- maxAggr(*) (with maxSD being a particular case) are heuristics that select directly a variable-value pair based on solution counting information without an explicit differentiation of variable and value ordering.
- minSC;maxSD, maxSC;maxSD, minT;maxSD, maxT;maxSD focus firstly on a specific constraint and then they select a variable-value pair among the variables in the preselected constraint scope.
- minDom;maxSD preselects firstly a subset of variables with minimum domain size and then chooses among them the one with the best variable-value pair according to counting information.
- max*Aggr*Var;maxSD differentiates the variable choice and the value choice (the previous heuristics choose directly a variable-value pair therefore they do not distinguish explicitly variable from value selection). Both selections are based on counting information.

A total of 23 different versions of heuristics are introduced, however it will be part of the experimental evaluation to shrink according to their performance the set of counting-based heuristics to only a few. All the heuristics assume a 2-way form of branching but they can easily be generalized to d-way.

Heuristic maxSD

The heuristic maxSD (Algorithm 5) simply iterates over all the variable-value pairs and chooses the one that has the highest density; assuming that the $\sigma(x_i, d, \gamma)$ are precomputed, the complexity of the algorithm is $O(qm)$ where q is the number of constraints and m is the sum of the cardinalities of the variables' domains. Interestingly, such a heuristic likely selects a variable with a small domain, in keeping with the *fail-first principle*, since its values have on average a higher density compared to a variable with many values (consider that the average density of a value is $\sigma(x_i, d, \gamma) = \frac{1}{|D_i|}$). Note that each constraint is considered individually.

```
1 max = 0;
2 for each constraint γ(x_1, ..., x_k) do
3     for each unbound variable x_i ∈ {x_1, ..., x_k} do
4         for each value d ∈ D_i do
5             if σ(x_i, d, γ) > max then
6                 (x*, d*) = (x_i, d);
7                 max = σ(x_i, d, γ);
8 return branching decision "x* = d*";
```

Algorithm 5: The Maximum Solution Density search heuristic (maxSD)

Heuristics maxAggr(*aggr*)

The heuristic maxAggr(*aggr*) (Algorithm 6) goes in the direction of aggregating, through simple functions, the counting information coming from different constraints. The algorithm iterates over each variable-value pair and it aggregates the solution densities. The aggregation function *aggr* denotes one of the following functions:

- max: $\max_{\gamma \in \Gamma(x_i)}(\sigma(x_i, d, \gamma))$ - selects the maximum of the solution densities. This is equivalent to the heuristic maxSD.
- maxRelSD: $\max_{\gamma \in \Gamma(x_i)}(\sigma(x_i, d, \gamma) - (1/|D_i|))$ - selects the maximum of the solution densities subtracting the average solution density for that given variable (i.e. $1/|D_i|$). maxAggr(maxRelSD) smoothes out the inherent solution densities differences due to domain cardinalities (as also the following aggregation function).
- maxRelRatio: $\max_{\gamma \in \Gamma(x_i)}(\frac{\sigma(x_i,d,\gamma)}{(1/|D_i|)})$ - selects the maximum of the ratio between the solution density and the average solution density for that given variable.
- min: $\min_{\gamma \in \Gamma(x_i)}(\sigma(x_i, d, \gamma))$ - selects the minimum of the solution densities. maxAggr(min) is a more conservative heuristics compared to maxAggr(max) as it branches on a variable-value pair that in the worst case still has a significant solution density.
- aAvg: $\frac{\sum_{\gamma \in \Gamma(x_i)} \sigma(x_i,d,\gamma)}{|\Gamma(x_i)|}$ - it computes the arithmetic average of the solution densities.
- wSCAvg: $\frac{\sum_{\gamma \in \Gamma(x_i)} (\#\gamma \sigma(x_i,d,\gamma))}{\sum_{\gamma \in \Gamma(x_i)} \#\gamma}$ - it computes the average of the solution densities weighted by the constraints' solution count. The weights tend to favor branchings on variable-value pairs that keep a high percentage of solutions on constraints with a high solution count.
- wAntiSCAvg: $\sum_{\gamma \in \Gamma(x_i)} \frac{(S_\gamma - \#\gamma)}{S_\gamma} \sigma(x_i, d, \gamma)$ - where $S_\gamma = \sum_{\gamma \in \Gamma(x_i)} \#\gamma$. It computes the average of the solution densities weighted by a function of the constraints' solution count. The weights tend to favor branchings on variable-value pairs that keep a high percentage of solutions on constraints with a low solution count.

- wTAvg: $\frac{\sum_{\gamma \in \Gamma(x_i)} (T_\gamma \sigma(x_i,d,\gamma))}{\sum_{\gamma \in \Gamma(x_i)} T_\gamma}$ - it computes the average of the solution densities weighted by the constraints' tightness.
- wAntiTAvg: $\sum_{\gamma \in \Gamma(x_i)} \frac{(S_T - T_\gamma)}{S_T} \sigma(x_i, d, \gamma)$ - where $S_T = \sum_{\gamma \in \Gamma(x_i)} T_\gamma$. It computes the average of the solution densities weighted by a function of the constraints' tightness. Weights tend to favor high solution density in constraints with low tightness.
- wDAvg: $\frac{\sum_{\gamma \in \Gamma(x_i)} D_\gamma \sigma(x_i,d,\gamma)}{\sum_{\gamma \in \Gamma(x_i)} D_\gamma}$ - it computes the average of the solution densities weighted by the cardinality of the Cartesian product of the domains of the variables in the constraint scope. The weights tend to preserve high solution densities in constraints that involve many other variable-value pairs.
- SCRemaining: $\sum_{\gamma \in \Gamma(x_i)} \#\gamma \, \sigma(x_i, d, \gamma)$ - it computes the remaining solution count of the constraints involving x_i, after branching on $x_i = d$ assuming that the propagations are performed independently.
- SCRemoved: $-\sum_{\gamma \in \Gamma(x_i)} \#\gamma \, (1 - \sigma(x_i, d, \gamma))$ - it computes the opposite of the number of constraint solutions removed, after branching on $x_i = d$ (assuming that the propagations are performed independently). Note that maxAggr(SCRemoved) is equivalent to minimizing the number of constraint solutions removed but it is *not* necessarily equivalent to maximizing the remaining constraint solutions maxAggr(SCRemaining).
- solProb: $\sum_{\gamma \in \Gamma(x_i)} T_\gamma \, \sigma(x_i, d, \gamma)$ - it computes the probability for a variable-value pair of being in a solution.
- avgSDDisc: $\frac{\sum_{\gamma \in \Gamma(x_i), \gamma' \in \Gamma(x_i) \,:\, \gamma \neq \gamma'} (|\sigma(x_i,d,\gamma) - \sigma(x_i,d,\gamma')|)}{|\Gamma(x_i)|(|\Gamma(x_i)|-1)}$ - it computes the average of the solution density discrepancy i.e. the average amount of the absolute differences between solution densities.
- maxSDDisc: $\max_{\gamma \in \Gamma(x_i), \gamma' \in \Gamma(x_i) \,:\, \gamma \neq \gamma'} (|\sigma(x_i, d, \gamma) - \sigma(x_i, d, \gamma')|)$ - it computes the maximum of the solution density discrepancy i.e. the maximum amount of the absolute difference between solution densities.

We tried to cover a wide variety of aggregation functions (and often also their opposite - *anti* heuristics): all of them are inspired either on the first-fail principle (i.e., branch on the subtree with the least solutions or the smallest probability of solutions) or on an optimistic succeed early on (i.e., branch on the subtree with the most solutions or the largest probability of solutions).

The first four aggregation functions still take the branching decision based on the information of a single constraint, whereas the following ones try to go in the direction of synthetizing a measure considering each constraint involved.

Of particular interest is maxAggr(avgSDDisc) and maxAggr(maxSDDisc) that try the most (compared to the other aggregation functions) to follow the *fail-first principle* as it branches

on the variable-value pair where the solution densities of the constraints disagree the most.

```
1 max = 0;
2 for each unbound variable x_i ∈ {x_1, ..., x_k} do
3     for each value d ∈ D_i do
4         aggrSD = aggr_{γ ∈ Γ(x_i)}(σ(x_i, d, γ));
5         if aggrSD > max then
6             (x*, d*) = (x_i, d);
7             max = aggrSD;
8 return branching decision "x* = d*";
```

Algorithm 6: The Maximum Aggregation search heuristic (maxAggr(aggr))

Assuming that the solution counting information is precomputed, the complexity of Algorithm 6 is $O(qm)$ except for maxAggr(avgSDDisc) for which the time complexity is $O(q^2m)$ since the aggregation function has a complexity of $O(q^2)$. In practice q is often smaller than the total number of problem constraints.

Heuristics minSC;maxSD, maxSC;maxSD, minT;maxSD, maxT;maxSD

Heuristic minSC;maxSD (Algorithm 7) first selects the constraint with the lowest number of solutions (line 2) and then iterates only over the variables involved in this constraint, choosing the variable-value pair with the highest solution density. The rationale behind this heuristic is that the constraint with the fewest solutions is probably among the hardest to satisfy. Assuming again that solution counting information is precomputed, the complexity of the algorithm is $O(q + |X(\gamma)|)$ where γ is the constraint selected.

```
1 max = 0;
2 choose constraint γ(x_1, ..., x_k) which minimizes #γ;
3 for each unbound variable x_i ∈ {x_1, ..., x_k} do
4     for each value d ∈ D_i do
5         if σ(x_i, d, γ) > max then
6             (x*, d*) = (x_i, d);
7             max = σ(x_i, d, γ);
8 return branching decision "x* = d*";
```

Algorithm 7: The Minimum Solution Count, Maximum Solution Density search heuristic (minSC;maxSD)

The maxSC;maxSD heuristic, preselects the constraint with the maximum solution count and then chooses the variable-value pair with the highest solution density. The minT;maxSD and maxT;maxSD are similar to the previous heuristics but they select the constraint with respectively the minimum and maximum tightness. These heuristics require the solution count of every constraint but then only need the solution density of variable-value pairs appearing in the chosen constraint. This too is advantageous if solution densities come at an additional computational price.

Heuristic minDom;maxSD

The heuristic minDom;maxSD (Algorithm 8) considers only the variables with the smallest domain size (line 2) and among these selects the variable-value pair with the highest solution density. In case only one variable has the smallest domain then the solution densities are exploited only for the value selection; on the other hand, if there is more than one variable with smallest domain, then counting-based information is involved also in the variable selection. The complexity of the algorithm is $O(qm)$.

```
1 max = 0;
2 Let S = {x_i : |D_i| > 1 and minimum};
3 for each variable x_i ∈ S do
4     for each constraint γ ∈ Γ(x_i) do
5         for each value d ∈ D_i do
6             if σ(x_i, d, γ) > max then
7                 (x*, d*) = (x_i, d);
8                 max = σ(x_i, d, γ);
9 return branching decision "x* = d*";
```

Algorithm 8: The Smallest Domain, Maximum Solution Density search heuristic (minDom;maxSD)

The above worst-case time complexity analysis is not necessarily indicative of the average behaviour of the heuristics. maxSD requires looking at every variable-value pair for every constraint but minDom;maxSD only needs to process the constraints that include the variables with the smallest domain and solution density information is only required for those variables. For this reason it is particularly suited for problems that involve constraints whose solution counting or solution density procedures are particularly time consuming.

Heuristics max*Aggr*Var;maxSD

So far the heuristics proposed blend together the variable and value selection heuristics. However it is possible to conceive heuristics that separate the variable selection from the value selection. For example, we might want to branch on a variable that has the highest average solution density no matter which value we choose (maxAvgVar;maxSD), or on variables for which there is a high regret (in terms of solution densities) in case we do not choose the best value (maxRegretVar;maxSD), or on a variable that has the best worst value (maxMinVar;maxSD). From preliminary tests, maxSD has proved to be one of the best (probably *the* best) counting-based heuristics therefore we decided to keep it as a value selection heuristic. Algorithm 9 shows the pseudo-code of the heuristic:

```
1 max = 0;
2 for each unbound variable x_i ∈ {x_1, ..., x_k} do
3     for each value d ∈ D_i do
4         maxSD_d = max_{γ ∈ Γ(x_i)}(σ(x_i, d, γ));
5     varScore_{x_i} = varAggr_{d∈D_i}(maxSD_d);
6 x* = argmax_{x_i∈{x_1,...,x_k}}(varScore_{x_i});
7 d* = argmax_{d∈D_{x*}}(maxSD_d);
8 return branching decision "x* = d*";
```

Algorithm 9: The Maximum Aggregation Variable: Maximum Solution Density search heuristic (max*Aggr*Var:maxSD)

For each value, the representative score is its maximum solution density (line 4). Clearly, this could possibly be changed to reflect any of the aggregation functions shown for the maxAggr(*aggr*) heuristic hence leading to a multitude of different heuristics. Once we ranked each value for a given variable, we can compute a variable score employing one of the following aggregation functions (denoted by varAggr in line 5 of the algorithm):

- Avg: $\frac{\sum_{d \in D_i} maxSD_d}{|D_i|}$ - it computes the arithmetic average of the value scores.
- Regret: $(maxSD_{d^*} - maxSD_{d^{**}})$ - it computes the difference between the best value score (d^*) and the second best value score (d^{**}).
- Max: $\max_{d \in D_i} maxSD_d$ - it returns the maximum of the value solution densities. This is equivalent to the maxSD heuristic therefore it is not used in the tests.
- Min: $\min_{d \in D_i} maxSD_d$ - it returns the minimum of the value scores.

The complexity of the Algorithm 9 is $O(qm)$.

4.3 Experimental Analysis

We performed a thorough experimental analysis (more than 6000 hours of running time) in order to evaluate the performance of the proposed heuristics on eight different problems described in the following sections. All the problems expose sub-structures that can be encapsulated in global constraints for which counting algorithms are known. Unfortunately, counting-based heuristics are of no use for random problems as this class of problems do not expose any structure; on the other hand, impact-based heuristics and conflict-driven heuristics can be employed also in random problems with very good results. This may be seen as a significant shortcoming for counting-based heuristics; nonetheless real-life problems usually do present structure therefore the performance of the heuristics proposed may have a positive impact in the quest of providing generic and efficient heuristics for structured problems. The problems that we tested vary significantly from one to the other, having different structures, different constraints with possibly different arities interconnected in different ways; thus, they can be considered as good representatives of problems that may arise in real life.

4.3.1 Experimental Settings

For all the problems, generalized arc consistency is maintained during search; the search tree is binary (i.e. $x_i = j \vee x_i \neq j$) and it is traversed through a depth-first search (unless specified differently). Domain consistency for `alldifferent`, `regular`, `knapsack` is maintained through the algorithms presented respectively in [88], [77], [104].

For what concerns counting-based heuristics, we employed the counting algorithms proposed in Chapter 3; when not specified the counting algorithm for the `alldifferent` constraint is UB-FC introduced in Section 3.1.3. The `knapsack` constraint is counted through an exact algorithm proposed in [79].

We tested the heuristics proposed in Section 4.2 and the following ones (see Section 4.1 as a reference):

- randomMinDom; randomVal - it selects among the variables with smallest remaining domain randomly and then chooses a value randomly
- brelaz; lex - it selects the variable according to the Brelaz heuristic and then chooses the value in lexicographic order.
- dom/ddeg; lex - it selects the variable according to the dom/ddeg heuristic and then chooses the value in lexicographic order.
- Ilog IBS - Impact-based Search with full initialization of the impacts.
- Ilog IBS+ - Impact-based Search with full initialization of the impacts; it chooses a subset of 5 variables with the best approximated and then it breaks ties based on the

node impacts while further ties are broken randomly.

- RSC+LA - Reduced Singleton Consistency + Look Ahead Heuristic
- RSC2+LA - Reduced Singleton Consistency on variables with smallest domain + Look Ahead Heuristic

Unfortunately we were not able to compare our results with the heuristic dom/wdeg as it is not obvious how to implement it in Ilog Solver (the solver we used): in fact, there is no access to the underlying propagation queue nor to the constraints that caused a failure, and explanations are not provided either. We did test it on a subset of problems that involve only constraints for which the constraint propagation code is available (hence allowing the detection of failures); this subset contains: Nonogram problem, Multi Knapsack and Market Split problem.

In order to figure out to which extent counting-based heuristics affect variable and value selection, we implemented some hybrid heuristics in which the variable is selected by one of the already known heuristics whereas value selection is delegated to maxSD. The hybrid heuristics are:

- brelaz; maxSD
- dom/ddeg; maxSD
- IBS; maxSD
- IBS+; maxSD

We were able to reproduce the results of Ilog IBS in our implementation (referred to as IBS); unfortunately that was not the case for Ilog IBS+. Therefore we report in the experimental analysis only IBS but both Ilog IBS+ and IBS+ (that is our implementation). The heuristics IBS; maxSD and IBS+; maxSD are based on our implementation of the impact-based heuristics.

All tests were performed on a AMD Opteron 2.2GHz with 1GB and Ilog Solver 6.6; the heuristics that involve some sort of randomization (either in the heuristic itself or in the counting algorithms employed) have been run 10 times and the average of the results has been taken into account. We set a timeout of 20 minutes for all the problems and heuristics.

We report in tables the following aggregated results:

- *a.T(S)* - arithmetic average of time for solved instances (in seconds)
- *a.T* - arithmetic average of time for solved and timeout instances (in seconds)
- *a.Bcks* - arithmetic average of number of backtracks for solved and timeout instances
- *g.T* - geometric average of time for solved and timeout instances (in seconds)
- *g.Bcks* - geometric average of number of backtracks for solved and timeout instances
- *% sol* - percentage of solved instances

N/A is reported when the result does not apply, or when numerical overflow occurred. Note

that except for *a.T(S)*, we accounted also timed out instances thus introducing a bias: however counting-based heuristics were frequently the one solving the highest percentage of instances therefore we expect to see increased gap in case we remove the timeout and let the solver run until it finds a solution. In order to compare more fairly the heuristics without depending from the chosen timeout, we also provide plots of % solved instances vs time for the significant heuristics. Finally, due to the large number of counting-based heuristics, we will report throughout the benchmarks only the heuristics with significant results. Full results can be found in Annex 6.

4.3.2 Quasigroup completion problem with holes

The Quasigroup completion problem with holes – QWH – (we will refer to this problem also as Latin Square problem) is defined on a $n \times n$ grid whose squares each contain an integer from 1 to n such that each integer appears exactly once per row and column (problem 3 of CSPLib [37]). The most common model uses a matrix of integer variables and an `alldifferent` constraint for each row and each column. We tested on the 40 hard instances used in [114] that have $n = 30$ and 42% of holes (corresponding to the phase transition) and were generated using [46]. Sixty additional instances outside the phase transition were generated to test the performance of the different counting algorithms on easy to medium difficulty instances (see Section 4.3.2). It is easily modeled as:

$$\begin{array}{ll} \texttt{alldifferent}((x_{i,j})_{1\leq j\leq n}) & 1 \leq i \leq n \\ \texttt{alldifferent}((x_{i,j})_{1\leq i\leq n}) & 1 \leq j \leq n \\ x_{i,j} = d & (i,j,d) \in S \\ x_{i,j} \in \{1,2,\ldots,n\} & 1 \leq i,j \leq n \end{array}$$

where S is the set of triplets indicating the preset cells (row, column, value). In this problem, each constraint is defined on n variables and is of the same type; each variable is involved in two constraints and has the same domain (disregarding the clues). This is a very homogeneous problem.

In Table 4.1, we report the results of the tested heuristics and in Figure 4.1 we plotted the percentage of solved instances vs time. Among the traditional heuristic the better performing ones are RSC+LA (being able to solve 95% of instances) and Ilog IBS+. The advantage of the former over the latter can be explained by the fact that QWH problems are particularly affected by the consistency level enforced: as we will describe in Chapter 5, QWH benefits significantly from Singleton Consistency. Only RSC+LA is able to solve a number of instances that is comparable to what counting-based heuristics can solve.

Analyzing hybrid heuristics, it is interesting to see how they largely benefit from the value

Table 4.1 Average results for 40 hard QWH instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	301.9	658.2	1300056	501.5	1001608	56.75%
Brélaz; lexico	254.0	750.6	1458276	432.6	846512	47.50%
dom/ddeg; lexico	333.8	723.6	1387456	404.9	780784	55.00%
IBS	125.1	716.3	900253	256.9	345228	45.00%
IBS+	335.0	532.9	1595756	273.9	826433	71.25%
Ilog IBS+	194.0	344.9	914849	91.1	247775	85.00%
RSC+LA	305.8	350.5	856	158.8	291	95.00%
RSC2+LA	125.8	179.5	4880	29.8	0	95.00%
Brélaz; maxSD	136.2	322.4	569170	38.1	65609	82.50%
dom/ddeg; maxSD	134.6	294.5	526034	46.7	85479	85.00%
IBS; maxSD	209.2	432.1	493791	154.8	188100	77.50%
IBS+; maxSD	207.1	430.5	493844	154.0	188111	77.50%
maxSD	77.1	105.2	104672	8.5	7512	97.50%
maxAggr(aAvg)	138.6	191.6	207699	19.1	19372	95.00%
maxAggr(wAntiTAvg)	87.8	199.0	177918	16.7	15239	90.00%
maxAggr(wDAvg)	141.8	194.7	195966	29.8	31511	95.00%
maxRegretVar; maxSD	102.8	130.2	159349	15.0	17815	97.50%

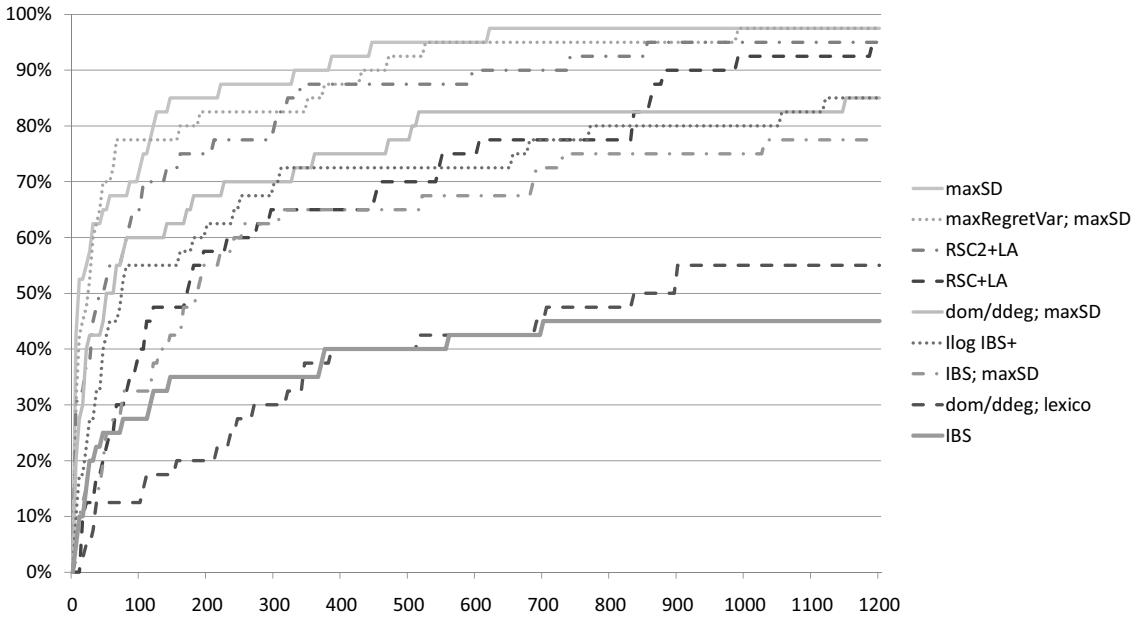


Figure 4.1 Percentage of solved instances vs time (in seconds) for QWH

selection based on solution densities: Brélaz; lexico and dom/ddeg; lexico both increase significantly the percentage of solved instances while at least halving solving times and number of backtracks. A similar gain can be seen also for impact based heuristics.

The best results come from counting-based heuristics, particularly with maxSD that has the lowest running time and the highest percentage of solved instances. They keep the lead also in term of the number of backtracks (two orders of magnitude when comparing the geometric averages) second only to RSC+LA that however subsumes a stronger consistency level.

Interestingly, the number of backtracks per second for counting-based heuristics is comparable to what we get with impact based heuristics meaning that the counting algorithm overhead is present but not overwhelming (the same cannot be said about RSC+LA that have a very low number of backtracks and thus a significant overhead); note that this measure impacts the eligibility of a heuristic to be employed with randomized restarts: an extremely low number of backtracks per second do not make a heuristic very suitable for restarts as it would be simply a waste of time.

It is worth observing that pure counting-based heuristics perform much better than hybrid ones, therefore confirming that counting-based heuristics are not only good value ordering heuristics but also effective variable ordering heuristic (this behavior is recurrent also on the next problems).

Adding Randomized Restarts

The QWH problem exhibits heavy-tail behavior in runtime distributions when the instances are generated close to the phase transition [44]. This means that there exists a strictly positive probability to reach a subtree during the search that requires exponentially more time to solve w.r.t. the other subtrees encountered so far. Nonetheless, heavy tails can be largely avoided by adding randomized restarts on top of the search procedure ([45]). This technique is orthogonal to the search heuristic employed and it systematically restarts the search every time a limit (typically a bound on the number of backtracks) is reached; obviously, in order to be effective, randomized restarts must be employed along with a heuristic that presents some sort of randomization such that at each restart different parts of the search tree are explored.

We tested a subset of our heuristics to assess their performance with randomized restarts. The heuristics tested are: Ilog IBS+, maxSD and rndMinSizeRndVal. The first two have been modified to guarantee randomness; particularly, one variable-value pair is chosen at random with equal probability between the best two provided by the heuristic. Note that, as pointed out in [86], impact information can be carried over different runs to improve the quality of

the impact approximation; no information is kept for the heuristic maxSD between restarts.

We implemented two universal strategies to generate the cutoff sequence of restarts: Luby [70] (i.e. $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, \dots$) and Walsh [109] (that is $1, r, r^2, \dots$ with $r = 2$); the scale parameter (i.e. the multiplier of the sequence elements) has been set to 5% of the number of variables after some pilot experiments.

Results are reported in Table 4.2 and Figure 4.2 shows the plot of the percentage of instances solved vs time.

Table 4.2 Average results for 40 hard QWH instances with randomized restarts

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal; rr. Luby	243.1	335.4	633163	152.0	291958	85.75%
rndMinSizeRndVal; rr. Walsh	271.2	458.6	881794	235.3	460519	74.75%
Ilog IBS+; rr. Luby	319.3	539.5	1679744	287.3	913213	75.00%
Ilog IBS+; rr. Walsh	196.4	497.5	1244723	166.5	430009	70.00%
maxSD; rr. Luby	40.4	46.1	41497	8.8	8084	98.25%
maxSD; rr. Walsh	56.9	59.6	57901	14.3	13700	99.25%

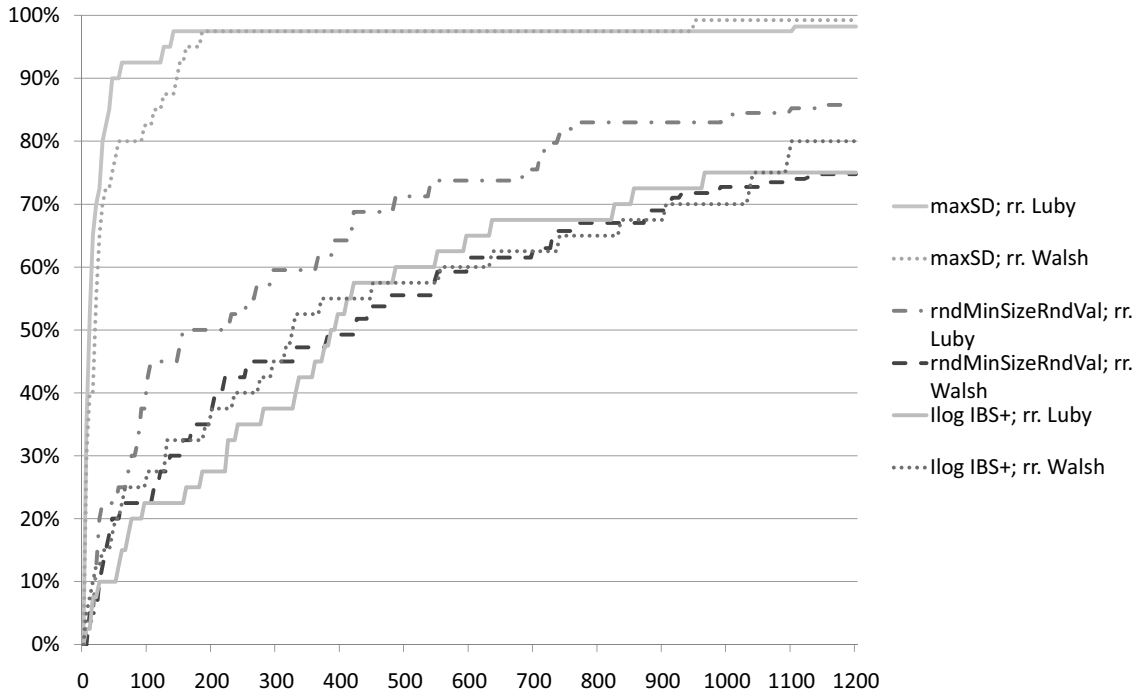


Figure 4.2 Percentage of solved instances vs time (in seconds) for QWH with randomized restarts

Randomized restarts sensibly improve the performance of rndMinSizeRndVal heuristic both

in terms of percentage of solved instances and time. Also maxSD with randomized restarts is able to cut considerably the solving time and the number of backtracks; after two minutes of running time it is able to solve 15% more instances compared to a pure maxSD. The heuristic Ilog IBS+ was not able to gain from the use of randomized restarts; however, note that heavy-tail behavior of runtime distribution is conjectured to depend on the problem structure *and* on the search heuristic employed (see [56]). We believe therefore that Ilog IBS+ is already limiting heavy tails for this particular problem thus rendering randomized restarts counterproductive.

Comparing alldifferent counting algorithms

In this section we compare the counting algorithms introduced in Chapter 3. To do so, we tested on the 40 hard instances used in the previous section (42% of holes) and we generated 60 additional instances outside the phase transition respectively with 45%, 47% and 50% of holes (using [46]) to see how much the counting algorithm overhead sums up in easy and hard instances. The counting algorithms tested are the following:

- UB-FC: based on upper bounds and enforcing forward checking during the local probes (the same used in the previous section)
- UB-AC: based on upper bounds and enforcing arc consistency during the local probes
- UB-DC: based on upper bounds and enforcing domain consistency during the local probes
- sampl-AC: based on sampling and enforcing arc consistency during sampling
- sampl-DC: based on sampling and enforcing domain consistency during sampling
- exact+sampl-AC: based on an initial exact enumeration and, in case of timeout, followed by sampling (enforcing arc consistency during sampling)
- exact+sampl-DC: based on an initial exact enumeration and, in case of timeout, followed by sampling (enforcing domain consistency during sampling)

For exact+sampl-AC and exact+sampl-DC, the exact enumeration have a timeout of 0.2 seconds. The advantage of a prior exact enumeration is twofold: it gives better guiding information w.r.t approximations; secondly, when close to the leaves of the search tree, that is, when the `alldifferent` are particularly constrained, exact enumeration may be sometimes faster than sampling. Nonetheless, it introduces a constant and significant overhead that pays off only if the exact enumeration succeeds; therefore, it is likely to be of limited use in easy instances where there is no necessity of high quality information. Furthermore, in our case, easy instances are particularly loose causing exact enumeration to fail in most cases.

The sampling phase does not have a timeout but it is instead bounded by the sample size, that is set dynamically. We got good results by setting the number of samples for

each constraint to ten times the maximum domain size of the variables in the scope of the constraint.

Table 4.3 Average solving time (in seconds) and number of backtracks for 100 QWH instances of order 30.

heuristic	a.T	a.B	%sol	a.T	a.B	%sol
	42% holes			45% holes		
Ilog IBS+	344.9	914849	85.00%	94.1	247556	95.00%
maxSD sampl-DC	398.8	15497	80.50%	20.0	619	100.00%
maxSD sampl-AC	339.7	15139	85.00%	29.0	1349	99.00%
maxSD exact+sampl-DC	132.0	4289	96.00%	115.2	517	99.5%
maxSD exact+sampl-AC	142.9	5013	97.00%	125.7	1092	98.5%
maxSD UB-DC	110.5	31999	95.00%	1.3	164	100.00%
maxSD UB-AC	82.4	68597	97.50%	0.7	582	100.00%
maxSD UB-FC	105.5	104496	97.50%	0.5	447	100.00%
	47% holes			50% holes		
Ilog IBS+	5.1	16126	100.00%	2.8	10012	100.00%
maxSD sampl-DC	22.8	657	99.00%	19.4	355	99.47%
maxSD sampl-AC	6.3	34	100.00%	7.7	8	100.00%
maxSD exact+sampl-DC	187.3	8	100.00%	269.0	29	100.00%
maxSD exact+sampl-AC	191.0	450	99.50%	262.0	2	100.00%
maxSD UB-DC	1.5	20	100.00%	2.4	3	100.00%
maxSD UB-AC	0.3	30	100.00%	0.3	2	100.00%
maxSD UB-FC	0.3	56	100.00%	0.3	6	100.00%

Results are shown in Table 4.3, for comparison we present also the results for the heuristic Ilog IBS+. Figures 4.3 and 4.4 show the percentage of solved instances within a given time for the instance sets with respectively 42% and 45% of holes (time is not cumulative). maxSD exact+sampl is the heuristic with the lowest number of backtracks on the hard instances together with a significantly lower runtime; however, as expected, it runs longer on the easy instances (see Figure 4.5): this can be explained by the fact that the easy instances have more loose constraints therefore the initial exact enumeration is more likely to time out. In Figure 4.3 we can see that the sampling algorithm alone is able to solve some instances within few seconds whereas maxSD exact+sampl-DC does not solve any instance within 40 seconds because of the high overhead due to exact enumeration. Sampling alone struggles more with the hard instances and it ends up solving just 85% of the instances whereas maxSD exact+sampl-DC solves 97% of the instances.The previous heuristics were significantly outperformed in all the instance sets by the heuristics based on upper bounds. As shown in Figure 4.5, maxSD UB-DC, maxSD UB-AC, maxSD UB-FC are very quick in solving easy instances and yet they are capable of solving the same number of instances as

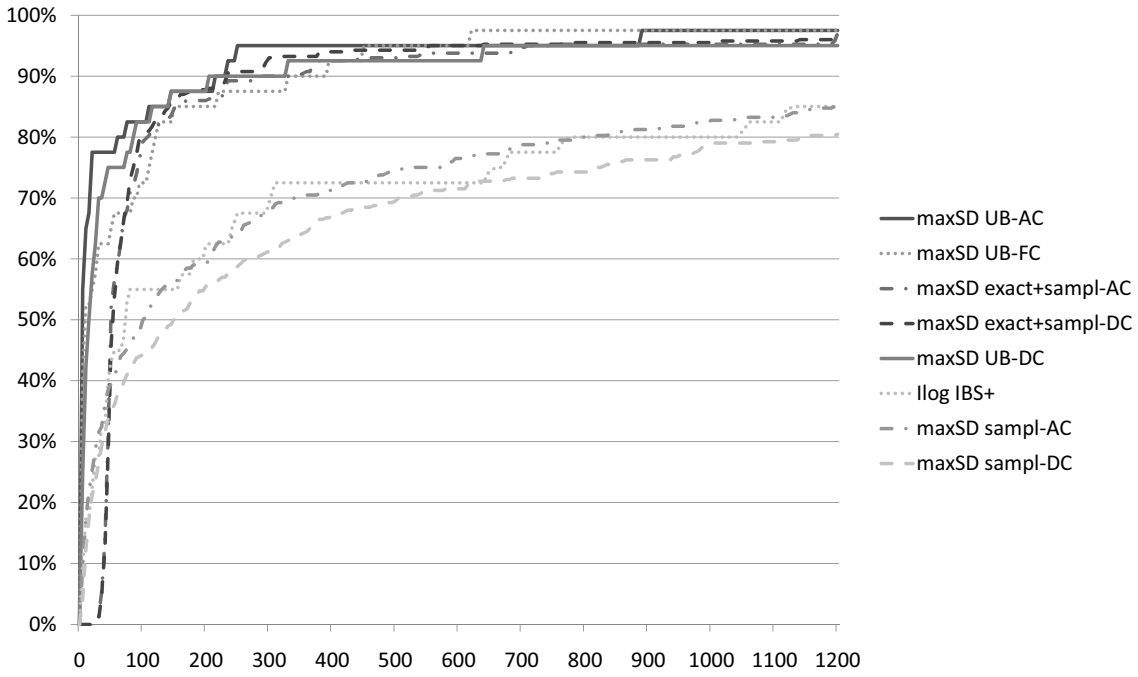


Figure 4.3 Percentage of solved instances vs time (in seconds) for QWH instances with 42% of holes

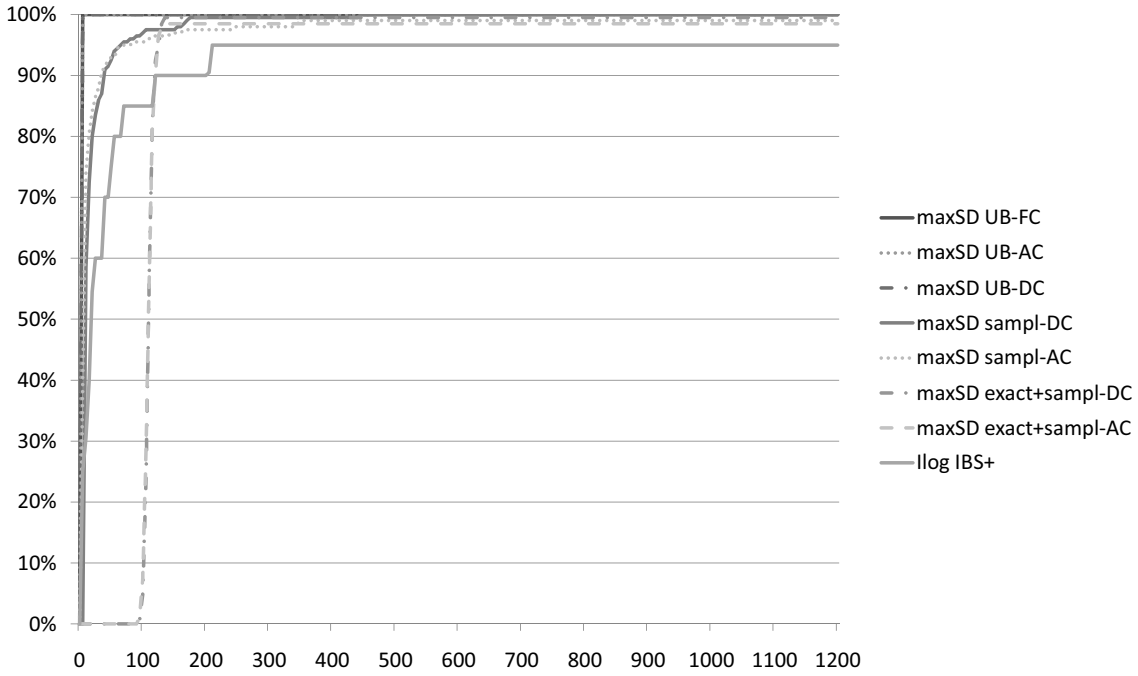


Figure 4.4 Percentage of solved instances vs time (in seconds) for QWH instances with 45% of holes

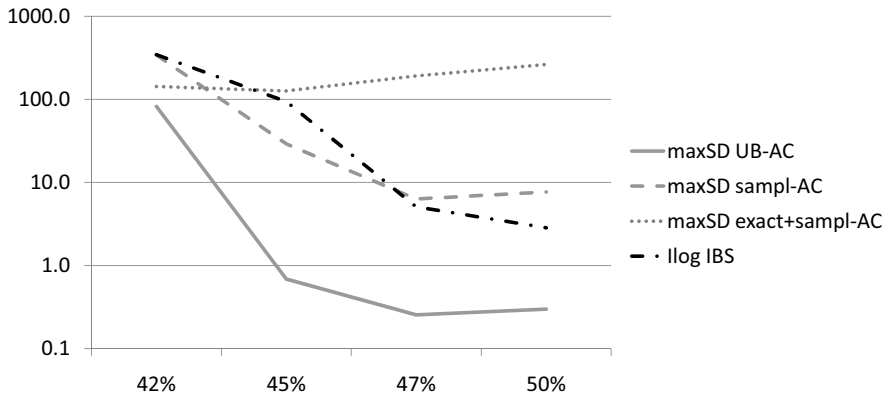


Figure 4.5 Total average time vs % holes

maxSD exact+sampl-DC. The latter heuristic shows its limit already in the set of instances with 45% of holes where no instance is solved within a hundred seconds, whilst maxSD based on upper bounds almost instantaneously solves all the instances. maxSD UB-AC was overall the best of the set on all the instances with up to a two orders of magnitude advantage over Ilog IBS+ in terms of solving time and up to four orders of magnitude for the number of backtracks. Enforcing a higher level of consistency leads to better approximated solution densities and to a lower number of backtracks, but it is more time consuming than simple arc consistency. A weaker level of consistency like forward checking can pay off on easy instances but it falls short compared to UB-AC on the hard ones. Note also that maxSD UB-DC increases the solving time, despite lowering the backtracks, when the instances have more holes (apart from the 42% holes instances): in those cases the sum of the domain cardinalities increases and the overhead of propagation becomes important. However we could not reuse the maximum matching and the strongly connected components (see [88]) computed for the propagation (there is no access to the underlying propagation code) — a more coupled integration of the counting algorithm with the propagation algorithm could lead to a performance gain.

4.3.3 Nonograms

A Nonogram (problem 12 of CSPLib [37]) is built on a rectangular $n \times m$ grid and requires filling in some of the squares in the unique feasible way according to some clues given on each row and column. As a reward, one gets a pretty monochromatic picture. Each individual clue indicates how many sequences of consecutive filled-in squares there are in the row (column), with their respective size in order of appearance. For example, "2 1 5" indicates that there are two consecutive filled-in squares, then an isolated one, and finally five consecutive ones. Each sequence is separated from the others by at least one blank square but we know little

about their actual position in the row (column). Such clues can be modeled with `regular` constraints (the actual automata $\mathcal{A}_i^r, \mathcal{A}_j^c$ are not difficult to derive):

$$\begin{array}{ll} \texttt{regular}((x_{i,j})_{1\leq j\leq m}, \mathcal{A}_i^r) & 1 \leq i \leq n \\ \texttt{regular}((x_{i,j})_{1\leq i\leq n}, \mathcal{A}_j^c) & 1 \leq j \leq m \\ x_{i,j} \in \{0,1\} & 1 \leq i \leq n,\ 1 \leq j \leq m \end{array}$$

This is a very homogeneous problem, with constraints of identical type defined over m or n variables, and with each (binary) variable involved in two constraints. These puzzles typically require some amount of search, despite the fact that domain consistency is maintained on each clue. We experimented with 180 instances [6] of sizes ranging from 16×16 to 32×32.

Table 4.4 Average results for 180 Nonogram instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	37.4	204.2	119756	0.0	0	84.83%
Brélaz; lexico	55.2	296.9	193618	0.0	0	78.89%
dom/ddeg; lexico	54.5	296.4	194889	0.0	0	78.89%
IBS	0.8	7.4	4712	0.0	0	99.44%
IBS+	5.1	6.9	12640	0.4	0	99.56%
llog IBS+	0.7	7.4	12668	0.0	0	99.44%
RSC+LA	2.9	2.9	10	0.5	0	100.00%
RSC2+LA	2.2	2.2	6	0.5	0	100.00%
Brélaz; maxSD	44.4	294.8	184267	0.0	0	78.33%
dom/ddeg; maxSD	44.9	295.2	186598	0.0	0	78.33%
IBS; maxSD	0.7	7.4	5785	0.0	0	99.44%
IBS+; maxSD	0.7	7.4	5707	0.0	0	99.44%
maxSD	8.7	48.5	18096	0.0	0	96.67%
maxAggr(wDAvg)	10.7	57.0	19782	0.0	0	96.11%
minAggr(SCRemoved)	4.6	37.8	13009	0.0	0	97.22%
maxAggr(avgSDDisc)	11.7	18.3	5312	0.0	0	99.44%
maxAvgVar; maxSD	3.9	17.2	5052	0.0	0	98.89%
maxMinVar; maxSD	6.5	26.4	7399	0.0	0	98.33%

Results are reported in table 4.4 and Figure 4.6 shows the percentage of solved instances vs time (in seconds). The majority of the instances are solved by all the heuristics within a second with only few taking more than 10 seconds. With a very limited number of backtracks, RSC+LA and RSC2+LA confirm to have a very low ratio of backtracks per second, nonetheless they are the only heuristics solving the whole benchmark set; this problem in fact is largely affected by the level of consistency achieved. Analogously, impact-based heuristics behave

6. Instances taken from http://www.blindchicken.com/~ali/games/puzzles.html

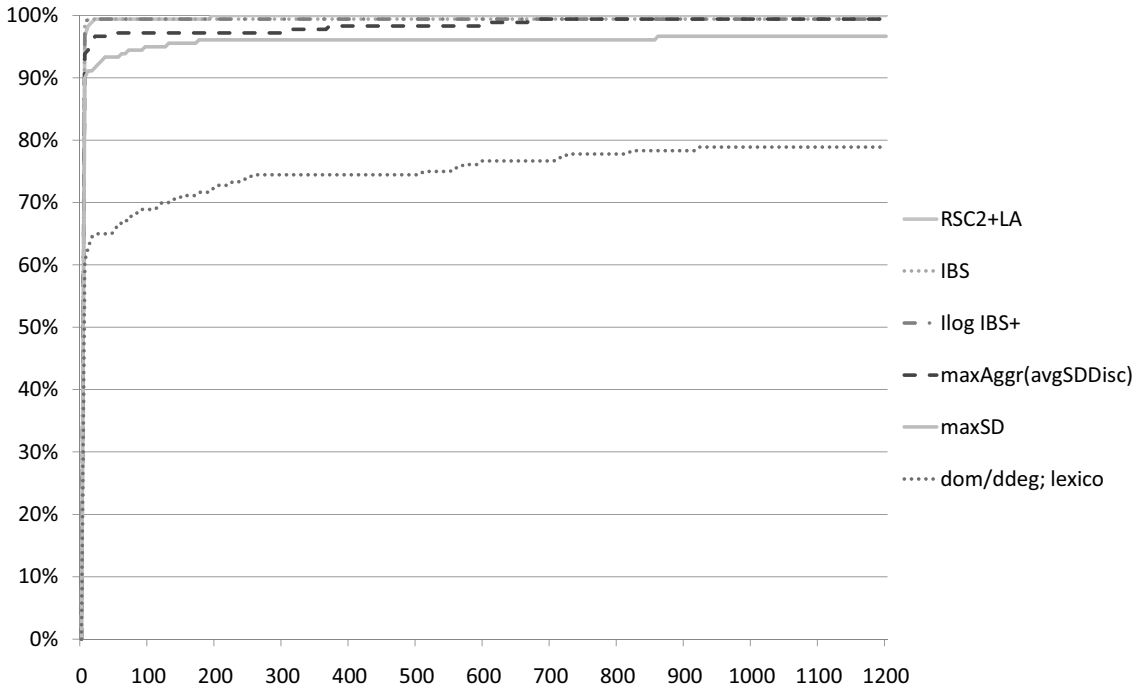


Figure 4.6 Percentage of solved instances vs time (in seconds) for the Nonogram problem

very well, partially thank to the full initialization of the impacts that subsumes a reduced form of singleton consistency at the root node. Heuristics such as Brélaz; lexico and dom/ddeg; lexico are the worse of the group; note that the two heuristics (and analogously Brélaz; maxSD and dom/ddeg; maxSD) boil down to be the same as the variables are binary and therefore variable ordering is solely based on the dynamic degree (differences on the number of backtracks is due to the timed out instances).

Here, counting-based heuristics struggle a bit more with only maxAggr(avgSDDisc) being able to solve the same amount of instances as of impact-based heuristics. The number of backtracks is however similar or lower than that of Ilog IBS+. The best results in term of solving time is obtained with maxAvgVar; maxSD that has an average solving time that is not too far from the best results got with traditional heuristics. Nonetheless, as the plot in Figure 4.6 shows, the difference between the best heuristic and the counting-based ones is not striking.

Hybridization does not bring any significant advantage to the traditional heuristics but at the same time it does not add any relevant overhead.

We report also another interesting result confirming the strong influence that the consistency level has on this problem: when applying at the root node the same reduced singleton consistency as of impact-based heuristics, maxSD was able to score a 97% of solved instances

with an average solving time of 2.4 seconds and with an average number of backtracks of less than 1600 (without considering timeout instances).

4.3.4 Multi dimensional knapsack problem

The Multi dimensional knapsack problem was originally proposed as an optimization problem by the OR community. We followed the same approach as in [86] in transforming the optimization problem into a feasibility problem by fixing the objective function to its optimal value, thereby introducing a 0-1 equality `knapsack` constraint. The other constraints are upper bounded `knapsack` constraints on the same variables. We tested on three different set of instances for a total of 25 instances: the first set corresponds to the six instances used in [86], the second set and the third set come from the OR-Library (Weish[1-13] from [96] and PB[1,2,4] and HP[1,2] from [30]). The first instance set have n, that is the number of variables, ranging from 6 to 50 and m, that is the number of constraints, from 5 to 10; in the second and third instance set n varies from 27 to 50 and m from 2 to 5. The problem has been modelled as follows:

$$
\begin{aligned}
& c^T x = c^* \\
& A^i x \leq u_i \qquad 1 \leq i \leq m \\
& x_j \in \{0,1\} \qquad 1 \leq j \leq n,
\end{aligned}
$$

where c is the cost vector, c^* stands for the optimal value, A is the matrix of the knapsack weights and with A^i we denote the $i-th$ row of the matrix. The problem involves only one kind of constraint and, differently from the previous problem classes, all the constraints are posted on the same set of variables.

Results are shown in Table 4.5 and, for the significant heuristics, in the plot of Figure 4.7. The pseudo-polynomial time complexity of the knapsack constraint propagation algorithm is reflected in the results where we can observe a very low number of backtracks per second throughout the heuristics.

As for the Nonogram problem, Brélaz and dom/ddeg coincide as variables are binary. Counting-based heuristics show a striking advantage in the percentage of instances solved being the only ones able to solve the totality of the benchmark set (traditional heuristics struggle to get more than three fourths of instances solved). The lead is kept also in running time that is one/two order of magnitude better than traditional heuristics and also in the number of backtracks that is substantially lower than the rest of the heuristics (between one and three orders of magnitude difference). Note that in this problem aggregation functions for maxAggr($\cdot$) such as the arithmetic average of the solution densities, bring some benefits. We believe that this might be explained by the fact that all the constraints share the same variables (in the Latin Square and Nonogram problems constraints overlap on only one

Table 4.5 Average results for 25 Multi Knapsack instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	169.3	620.9	11585	90.6	1518	51.60%
Brélaz; lexico	102.4	497.9	11212	0.0	0	64.00%
dom/ddeg; lexico	104.9	499.5	11130	0.0	0	64.00%
IBS	86.5	354.0	4080	41.6	679	76.00%
IBS+	258.9	435.0	8607	72.8	1654	75.20%
Ilog IBS+	140.7	395.6	4121	63.2	1256	76.00%
RSC+LA	204.6	603.1	450	0.0	0	60.00%
RSC2+LA	230.1	618.3	527	144.2	0	60.00%
Brélaz; maxSD	111.9	504.1	9948	0.0	0	64.00%
dom/ddeg; maxSD	107.1	501.0	9877	50.4	0	64.00%
IBS; maxSD	163.7	412.8	3179	0.0	768	76.00%
IBS+; maxSD	164.4	413.2	3242	0.0	770	76.00%
maxSD	25.7	72.7	1228	0.0	0	96.00%
maxAggr(aAvg)	6.7	6.7	29	0.0	0	100.00%
maxAggr(wAntiSCAvg)	9.4	9.4	509	0.0	0	100.00%
maxAggr(wAntiTAvg)	9.9	9.9	509	0.0	0	100.00%
maxAggr(wDAvg)	6.7	6.7	29	0.0	0	100.00%
maxRegretVar; maxSD	71.8	71.8	2880	0.0	0	100.00%

variable); therefore branching while considering all the constraint informations pays off.

We got mixed results for what concerns the hybrid heuristics: Brélaz and dom/ddeg do not report any advantage nor performance degradation; IBS somewhat looses in time and IBS+ gains from hybridization. Note however that generally they are all able to decrease the number of backtracks. Nonetheless, hybrid heuristics remain very far from pure counting-based heuristic, underlining once more how these latter can play a crucial role as variable ordering heuristics.

For the sake of clarity, in this benchmark set, impact-based heuristics are partially limited by the full initialization of the impacts (the same element that makes the heuristic very effective in other benchmarks). We recall that this procedure, aside from the impact initialization, subsumes also a reduced form of singleton consistency (at the root node); this aspect proved to bring advantages for example in the Nonogram problem. However, when the propagation is particularly heavy-weight (as in this problem), probing the impact for each variable-value pair may take a significant amount of time (despite the relatively low number of variable-value pairs). This overhead has been measured to be of about 29 seconds.

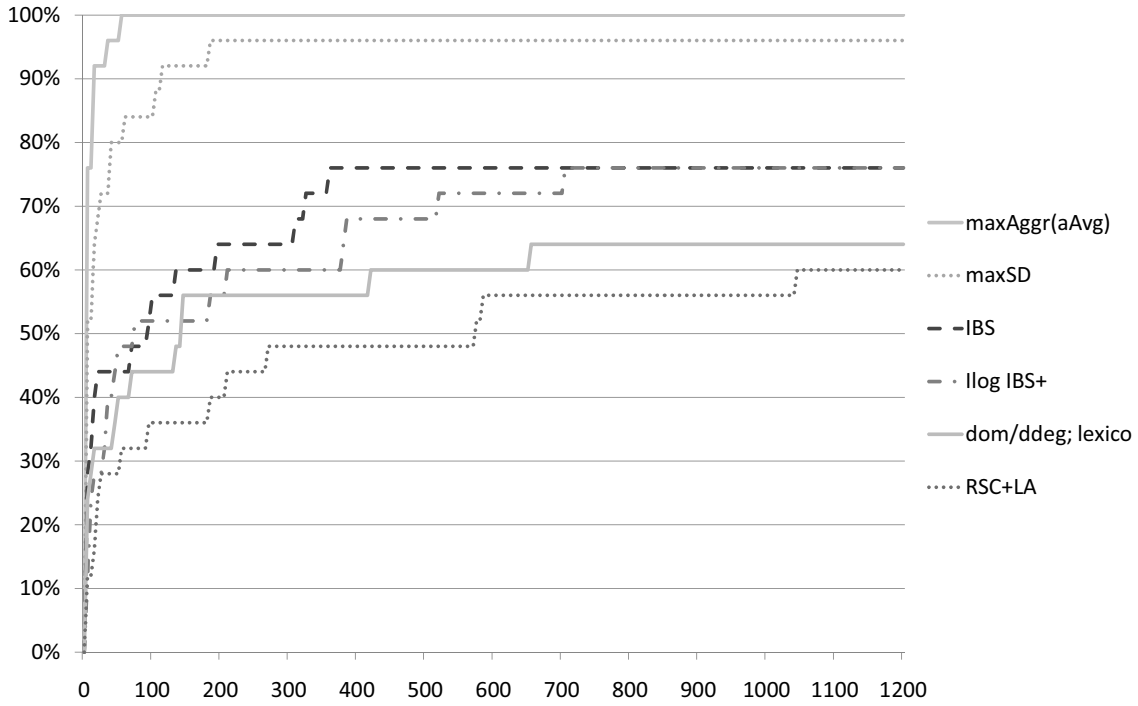


Figure 4.7 Percentage of instances solved vs time (in seconds) for the Multi Knapsack problem

4.3.5 Market Split Problem

The Market Split Problem was originally introduced by [21] as a challenge to LP-based branch-and-bound approaches. There exists both a feasibility and optimization version. The feasibility problem consists of m 0-1 equality `knapsack` constraints defined on the same set of $10(m-1)$ variables. Even small instances ($4 \leq m \leq 6$) are surprisingly hard to solve by standard means. We used the 10 instances tested in [79] that were generated with [110]. The problem has been modelled as follows:

$$A^i x = b_i \qquad 1 \leq i \leq m$$
$$x_j \in \{0,1\} \qquad 1 \leq j \leq n,$$

Although apparently very similar to the Multi Dimensional Knapsack problem, the Market Split turns out to be much tougher to solve in practice. It shares however the same characteristics: the constraints are of the same type and they are posted on the same set of variables.

Results are shown in Table 4.6 and, for the significant heuristics, in Figure 4.8. The first observation that sets this problem apart from the others, is the number of backtracks throughout the heuristics: the difference between the less informed heuristics (Brélaz; lexico and dom/ddeg; lexico for instance) and the more informed (impact-based or counting-based

Table 4.6 Average results for 10 Market Split instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	268.7	268.7	115538	257.1	110044	100.00%
Brélaz; lexico	121.6	121.6	109742	101.9	91070	100.00%
dom/ddeg; lexico	122.4	122.4	109742	102.9	91070	100.00%
IBS	202.3	202.3	90936	162.9	76601	100.00%
IBS+	584.5	688.0	358098	670.3	345023	83.00%
Ilog IBS+	620.9	736.8	420780	599.0	327518	80.00%
RSC+LA	716.3	909.8	22178	843.5	20208	60.00%
RSC2+LA	723.3	914.0	23158	849.5	21129	60.00%
Brélaz; maxSD	245.9	245.9	93213	141.2	53613	100.00%
dom/ddeg; maxSD	231.8	231.8	93213	136.5	53613	100.00%
IBS; maxSD	328.6	328.6	89893	184.2	52235	100.00%
IBS+; maxSD	338.6	338.6	89893	187.6	52235	100.00%
maxSD	256.9	256.9	59878	182.0	40944	100.00%
maxAggr(aAvg)	274.8	274.8	69251	176.5	43986	100.00%
maxAggr(avgSDDisc)	175.8	175.8	37852	109.5	22928	100.00%
minSCMaxSD	210.6	210.6	50944	167.4	39812	100.00%
minTMaxSD	209.6	209.6	50944	164.8	39812	100.00%

heuristics) is not as marked as in the other problems (same order of magnitude or at most a factor of two of difference).

Solving times reflect this behavior: the heuristic with the least overhead are the one with the best running time (Brélaz; lexico and dom/ddeg; lexico - note that the two are again identical as the variables are binary); in this respect impact-based and counting-based heuristics lag behind. RSC+LA and RSC2+LA overhead is a strong limitation in this problem as these heuristics are not able to solve the whole instance set.

The number of backtracks of counting-based heuristics is the lowest among the heuristics enforcing the same consistency level but the added overhead of counting algorithms does not pay off when compared to simple heuristics such as dom/ddeg; lexico. Interestingly, maxAggr(avgSDDisc), strongly inspired by the *first-fail* principle, has the lowest number of backtracks having also a running time close to the best heuristics.

Finally, hybrid heuristics in this problem do not improve over their pure heuristic counterpart (except for IBS+; maxSD): indeed, they are able to slightly descrease the number of backtracks, nonetheless this is not sufficient to make up the additional overhead due to counting algorithms.

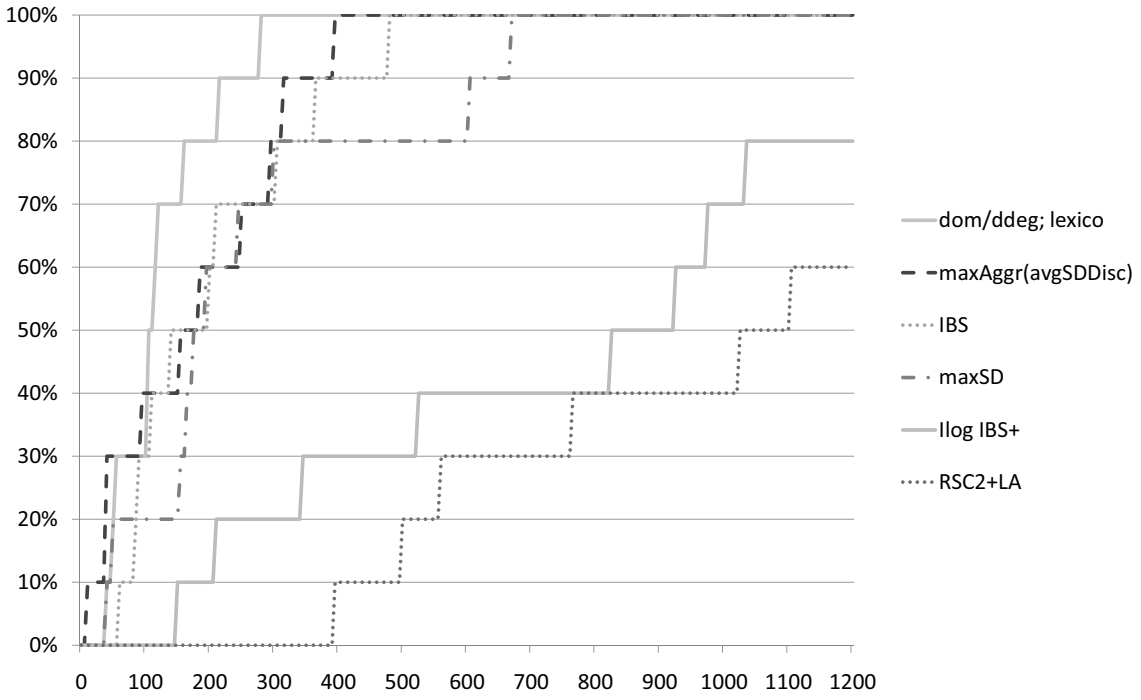


Figure 4.8 Percentage of instances solved vs time (in seconds) for the Market Split problem

4.3.6 Magic Square Completion Problem

This puzzle (problem 19 of the CSPLib [37]) is defined on a $n \times n$ grid and asks to fill the square with the number from 1 to n^2 such that each row, each column and each main diagonal sums up to the same value. In order to make them harder, the problem instances have been partially prefilled (half of the instances have 10% of the variables set and the remaining 50% of the variables set). The 40 instances (9×9) are taken from [79]. This problem is modelled with a matrix of integer variables, a single `alldifferent` constraint spanning over all the variables and a `knapsack` constraint for each row, column and main diagonal:

$$\begin{aligned}
&\textstyle\sum_{i=1}^{n} x_{i,j} = S && 1 \leq j \leq n \\
&\textstyle\sum_{j=1}^{n} x_{i,j} = S && 1 \leq i \leq n \\
&\textstyle\sum_{i=1}^{n} x_{i,i} = S \\
&\textstyle\sum_{i=1}^{n} x_{i,n-i-1} = S \\
&\texttt{alldifferent}((x_{i,j})_{1 \leq j \leq n, 1 \leq i \leq n}) \\
&x_{i,j} \in \{1, n^2\} && 1 \leq i \leq n, 1 \leq j \leq n,
\end{aligned}$$

where $S = n(n^2 + 1)/2$. The problem involves different constraints although the majority are equality `knapsack` with the same arity.

We report the results in Table 4.7 and plot the percentage of solved instances vs time

Table 4.7 Average results for 40 Magic Square instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	59.4	140.4	29665	72.0	14436	93.00%
Brélaz; lexico	58.9	658.7	72877	152.0	14438	47.50%
dom/ddeg; lexico	36.5	736.0	83113	197.8	21024	40.00%
IBS	73.1	638.5	24447	279.3	12679	50.00%
IBS+	644.0	716.3	43107	499.0	28449	72.00%
Ilog IBS+	551.0	632.4	20471	357.2	11658	85.25%
RSC+LA	626.8	985.7	39	926.3	0	37.50%
RSC2+LA	219.4	734.9	897	420.5	0	47.50%
Brélaz; maxSD	34.3	34.3	10592	6.3	394	100.00%
dom/ddeg; maxSD	38.3	38.3	11695	6.2	473	100.00%
IBS; maxSD	296.7	296.7	5282	178.4	4643	100.00%
IBS+; maxSD	302.2	302.2	5282	182.5	4643	100.00%
maxSD	14.1	14.1	1685	8.1	203	100.00%
maxAggr(wAntiSCAvg)	11.9	41.6	11998	8.4	273	97.50%
maxAggr(wAntiTAvg)	10.5	10.5	1290	6.0	98	100.00%
maxAggr(avgSDDisc)	66.5	123.2	30350	21.5	1013	95.00%
maxAvgVar; maxSD	24.6	24.6	3564	8.7	114	100.00%

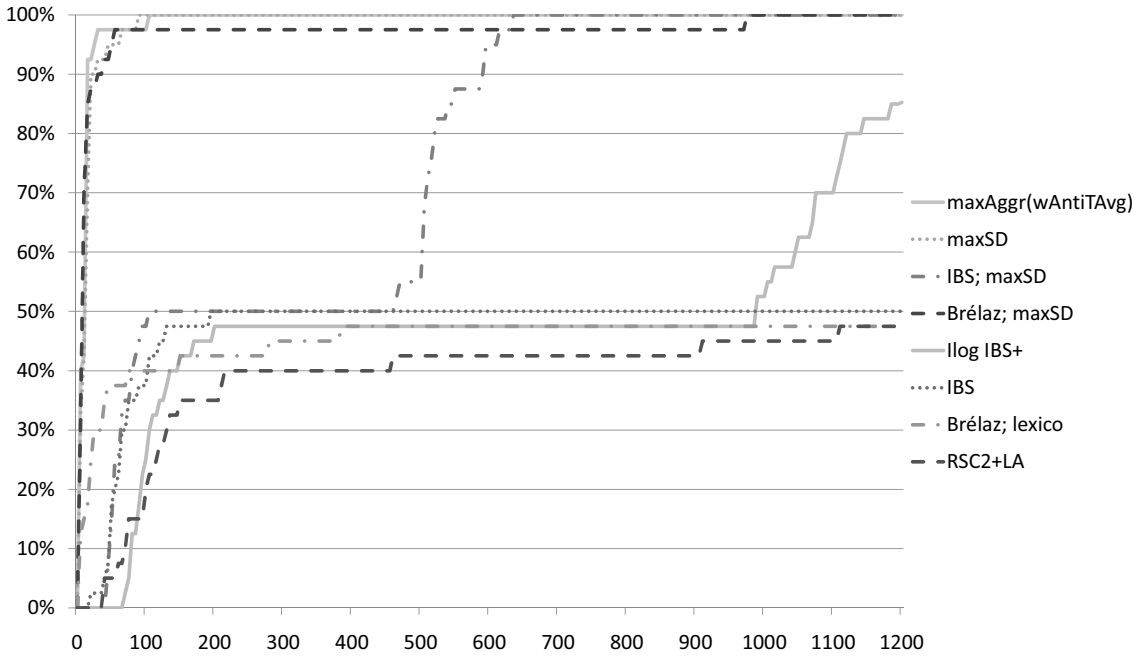


Figure 4.9 Percentage of instances solved vs time (in seconds) for the Magic Square problem

in Figure 4.9. In this problem, easier instances (50% of prefilled cells) are solved by the majority of the heuristics, however the ones prefilled at 10% present a real challenge for the traditional heuristics. Among this group, the only heuristic having a decent running time while being able to solve almost all the instances is the rndMinSizeRndVal heuristic; for the remaining, the running time is significantly high. The heuristics based on reduced singleton consistency are the worse of the group, as they cannot solve more than half of the instances.

Applying a value selection based on solution densities improves considerably the performance as it manages to decrease the number of backtracks by a factor of 4 to 8 (see Figure 4.9); furthermore, hybrid heuristics are able to solve all of the instance set.

Counting-based heuristics have a clear lead on this kind of problem with maxSD and maxAggr(wAntiTAvg) being the best heuristics in term of running times and percentage of instances solved. Geometric average of backtracks is up to two orders of magnitude less than traditional heuristics.

Finally, we remark that, as in the Multi Knapsack problem, the full impact initialization is a very heavy procedure for this problem due to the high number of variable-value pairs to probe ($\approx n^4$ that is in our instances $9^4 = 6561$) and the presence of constraint propagators whose running time is pseudo-polynomial. The overhead has been measured to be of about 4 minutes and 45 seconds.

4.3.7 Cost-Constrained Rostering Problem

This problem has been borrowed from [79] and the 10 instances as well. It is inspired by a rostering problem where m employees ($m = 4$) have to accomplish a set of tasks in a n-days schedule ($n = 25$). No employee can perform the same task of another employee on the same day (`alldifferent` constraint on each day). Moreover, there is an hourly cost for making someone work, which varies both across employees and days. For each employee, the total cost must be equal to a randomly generated value (equality `knapsack` constraint for each employee). Finally, each instance has about 10 forbidden shifts i.e. there are some days in which an employee cannot perform a given task. In the following, we refer to this problem also as KPRostering. The problem can be modelled as follows:

$$
\begin{array}{ll}
\sum_{j=1}^{n} c_{i,j} x_{i,j} = b_i & 1 \leq i \leq m \\
\texttt{alldifferent}((x_{i,j})_{1 \leq i \leq n}) & 1 \leq j \leq n \\
x_{i,j} \neq d & (i,j,d) \in F \\
x_{i,j} \in \{1, m\} & 1 \leq i \leq m, 1 \leq j \leq n,
\end{array}
$$

where F is the set of forbidden combinations. This problem presents constraints of different types that have largely different arities.

Results are shown in Table 4.8 and in Figure 4.10. This problem exhibits a significant

Table 4.8 Average results for 10 Cost-Constrained Rostering instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	128.7	756.4	722226	88.5	53973	39.00%
Brélaz; lexico	158.7	679.4	995657	46.1	0	50.00%
dom/ddeg; lexico	162.8	681.4	1004048	42.6	0	50.00%
IBS	51.3	510.8	264037	62.9	31071	60.00%
IBS+	155.3	745.1	898336	206.5	258157	41.00%
Ilog IBS+	39.1	503.4	616198	51.2	50695	60.00%
RSC+LA	434.4	1046.9	8386	796.8	6134	20.00%
RSC2+LA	4.5	482.7	33408	37.4	0	60.00%
Brélaz; maxSD	110.0	655.0	743228	66.4	33530	50.00%
dom/ddeg; maxSD	114.9	657.5	748973	65.1	33769	50.00%
IBS; maxSD	13.5	488.1	233674	43.6	20485	60.00%
IBS+; maxSD	13.5	488.1	232443	43.8	20491	60.00%
maxSD	0.3	0.3	5	0.3	0	100.00%
maxAggr(maxRelSD)	0.3	0.3	0	0.3	0	100.00%
maxAggr(wAntiTAvg)	0.3	0.3	12	0.3	0	100.00%
maxAggr(wDAvg)	0.3	0.3	2	0.3	0	100.00%
maxRegretVar; maxSD	0.3	0.3	0	0.3	0	100.00%

difference in performance across the heuristics. No traditional heuristics nor hybrid ones are able to solve all of the instances. Only a few can solve 60% of the instances and the best one for what concerns running time is RSC2+LA.

Hybrid heuristics are able to improve over the traditional ones with respect to running time, however they are still limited to 50% - 60% of solved instances.

Counting-based heuristics show a significant advantage over all the other heuristics. Many variations of them solve all the instances of this benchmark set.

Even when considering only solved instances, the running time of counting-based heuristics can be up to three orders of magnitude less than traditional heuristics. The number of backtracks is very small; for maxAggr(maxRelSD) and maxRegretVar; maxSD the search is even backtrack-free across all the 10 instances.

Once more it should be noted that pure counting-based heuristics make a difference for what concerns variable ordering; the driving force in this benchmark set comes largely from the knapsack constraints (consider that these span over 25 variables whereas `alldifferent` constraints are limited to 4 variables).

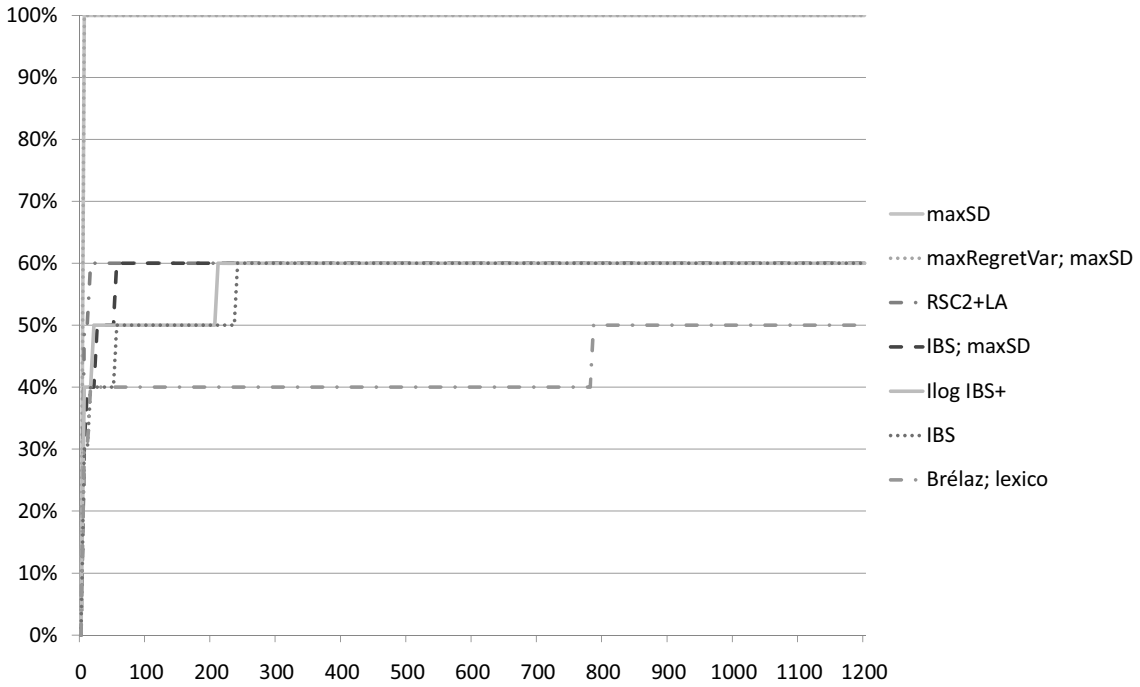


Figure 4.10 Percentage of instances solved vs time (in seconds) for Cost-Constrained Rostering problem

4.3.8 Rostering Problem

This problem has been inspired by a rostering context as the previous problem, however it has different caracteristics. Here, the objective is to schedule n employees over a span of n time period. In each time period, $n-1$ tasks need to be accomplished and one employee out of the n has a break. The tasks are fully ordered 1 to $n-1$; for each employee the schedule has to respect the following rules: two consecutive time periods have to be assigned to either two consecutive tasks (in no matter which order i.e. $(t, t+1)$ or $(t+1, t)$) or to the same task (i.e. (t, t)); there cannot be more than three consecutive time periods assigned to the same task; an employee can have a break after no matter which task; after a break an employee cannot perform the task that preceeds the task prior to the break (i.e. $(t, P, t-1)$ is not allowed).
We generated 2 sets of 30 instances with $n = 10$ each with a 5% of preset assignment and respectively 0% and 0.025% of values removed. The model employed is the following:

$$
\begin{array}{ll}
\texttt{regular}((x_{ij})_{1\leq j\leq n}, FP) & 1 \leq i \leq n \\
\texttt{alldifferent}((x_{ij})_{1\leq i\leq n}) & 1 \leq j \leq n \\
x_{i,j} \neq d & (i,j,d) \in F \\
x_{i,j} = d & (i,j,d) \in S \\
x_{i,j} \in \{1,\ldots,n\} & 1 \leq i \leq n, 1 \leq j \leq n,
\end{array}
$$

where S is the set of preset variables, F the set of forbidden variable-value assignments and FP defines the set of forbidden patterns as described above. A variable assignment $x_{i,j} = d$ (with $d = 1,\ldots,n-1$) means that employee i performs task d in time period j; the value n is reserved for the break. Note that the rules defined on each employee schedule have been modelled via a regular constraint that embeds both the stretch and the pattern constraint (see [77]).

Table 4.9 Average results for 60 Rostering instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	12.0	231.0	2179673	25.9	0	81.50%
Brélaz; lexico	0.3	120.3	1302799	0.1	0	90.00%
dom/ddeg; lexico	0.3	120.3	1318698	0.0	0	90.00%
IBS	0.9	0.9	856	0.9	853	100.00%
IBS+	1.3	1.3	1838	1.3	1836	100.00%
Ilog IBS+	1.3	1.3	1854	1.2	1851	100.00%
RSC+LA	10.0	10.0	0	9.4	0	100.00%
RSC2+LA	159.0	853.0	220248	143.8	0	33.33%
Brélaz; maxSD	150.1	937.5	6647932	292.9	0	25.00%
dom/ddeg; maxSD	148.1	954.6	6770603	337.4	N/A	23.33%
IBS; maxSD	0.9	0.9	868	0.9	864	100.00%
IBS+; maxSD	0.9	0.9	868	0.9	864	100.00%
maxSD	5.6	164.8	416609	0.7	0	86.67%
maxAggr(avgSDDisc)	0.1	0.1	0	0.1	0	100.00%
maxAggr(maxSDDisc)	0.1	0.1	0	0.1	0	100.00%
maxRegretVar; maxSD	0.1	0.1	0	0.1	0	100.00%

Results are reported in Table 4.9 and Figure 4.11. This problem turns out to be relatively easy to solve with only the heuristic RSC2+LA struggling to solve all the instances. Hybrid heuristics here show a negative result particularly for Brélaz; maxSD and dom/ddeg; maxSD where the value selection is completely off when compared to a pure lexicographic ordering. Note however that the specific rules of this rostering problem back up very well a lexicographic value selection. The same behavior appears also in the pure maxSD heuristic that lags behind compared to all the traditional heuristics as it is not able to solve all of the benchmark set.

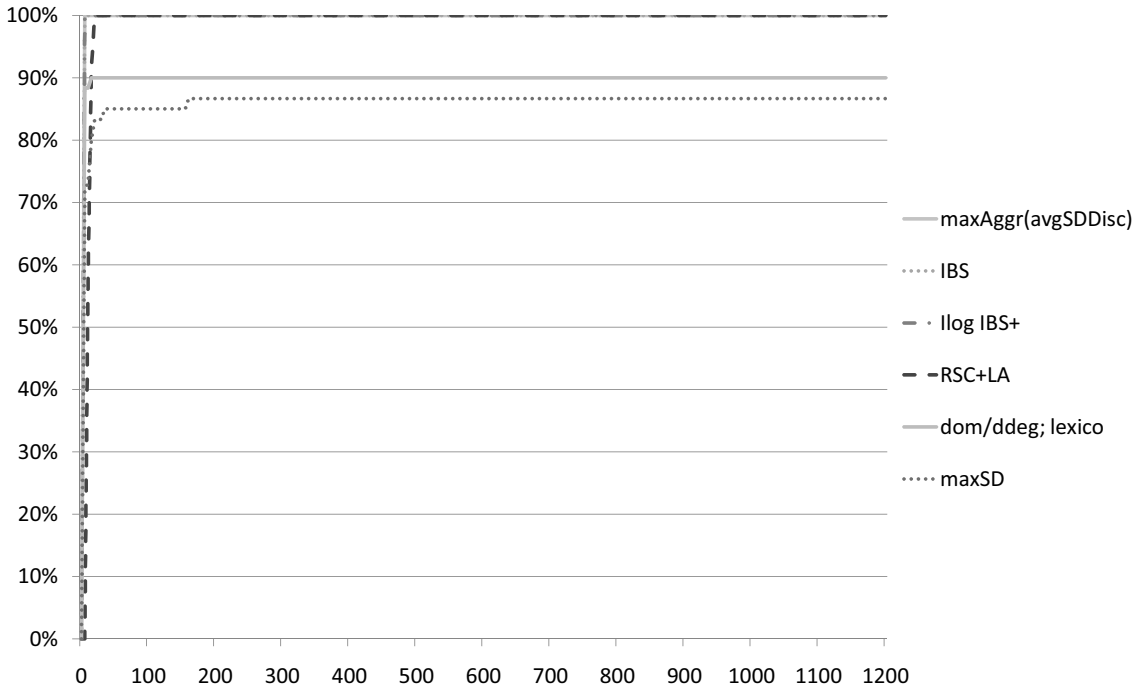


Figure 4.11 Percentage of instances solved vs time (in seconds) for Rostering problem

This is reflected also in the number of backtracks sensibly higher than, for example, impact-based heuristics. Other counting-based heuristics however showed very good results providing solutions in a backtrack-free fashion. We believe that also in this problem focusing on a single constraint (such as in MAXSD) misses the chance of capturing the strongly interwoven effect of the constraint problems.

4.3.9 Travelling Tournament Problem with Predefined Venues

The Travelling Tournament Problem With Predefined Venues (TTPPV) has been introduced in [74] and it consists of finding an optimal single round robin schedule for a sport event. Given a set of n teams, each team has to play against each other team. In each game, a team is supposed to play either at home or away, however no team can play more than three consecutive times at home or away. The particularity of this problem resides on the venues of each game that is predefined, i.e. if team a plays against b it is already known whether the game is going to be held at a's home or at b's home. A TTPPV instance is said to be balanced if the number of home games and the number of away games differ by at most one for each team; otherwise it is referred to as non-balanced or random.

The TTPPV was originally introduced as an optimization problem where the sum of the travelling distance of each team has to be minimized, however [74] shows that it is particularly

difficult to find a single feasible solution employing traditional integer linear programming methods. Balanced instances of size 18 and 20 (the number of teams denotes the instance size) were taking from roughly 20 to 60 seconds to find a first feasible solution with Integer Linear Programming; non-balanced instances could take up to 5 minutes (or even time out after 2 hours of computation); furthermore, six non-balanced instances are infeasible with ILP approach proposed in [74] unable to prove it. Hence, the feasibility version of this problem already represents a challenge. We propose two equivalent models; the first one, referred to as TTPPV1, is the following:

$$
\begin{array}{ll}
x_{i,j} = k \iff x_{k,j} = i & 1 \leq i \leq n, 1 \leq j \leq n-1 \\
x_{i,j} \neq i & 1 \leq i \leq n, 1 \leq j \leq n-1 \\
\texttt{alldifferent}((x_{i,j})_{1 \leq j \leq n-1}) & 1 \leq i \leq n \\
\texttt{regular}((ha_{i,j})_{1 \leq j \leq n-1}, PV_i) & 1 \leq i \leq n \\
\texttt{alldifferent}((x_{i,j})_{1 \leq i \leq n}) & 1 \leq j \leq n-1 \\
x_{i,j} = k \Rightarrow ha_{i,j} = PV_{i,k} & 1 \leq i \leq n, 1 \leq j \leq n-1 \\
ha_{i,j} \in \{0, \ldots, 1\} & 1 \leq i \leq n, 1 \leq j \leq n-1, \\
x_{i,j} \in \{1, \ldots, n\} & 1 \leq i \leq n, 1 \leq j \leq n-1,
\end{array}
$$

A variable $x_{i,j} = k$ means that team i plays against team k at round j; consistency among these variables is maintained through the first set of constraints (i.e. if a plays against b in round j then b plays against a in the same round). The second set of constraints forbids degenerate solutions in which a team plays against himself. The `alldifferent` constraints defined on the rows enforce that each team plays against every other team. The home-away pattern is defined through a `regular` constraint (defined through a slightly overloaded notation) on a matrix of auxiliary binary variables ha (as in traditional models of the Travelling Tournament Problem). Those variables depend directly on the value of the variables x and on the set of predefined venues PV (with $PV_{i,k} = 0 \iff PV_{k,i} = 1$ when team i plays at home against team k). Finally the `alldifferent` constraint on the columns is redundant and employed for achieving additional filtering. Note that the first set of constraints and the `alldifferent` constraint could have been replaced by a set of `symmetric_alldifferent` constraints ([87]).

In this first model, branching variables are the x variables, and the only counting information available comes from the `alldifferent` constraints; therefore counting-based heuristics branch without considering the home-away pattern. Note that another possibility is to branch indistinctly on x and ha variables with counting information coming from `alldifferent` constraints for the first and from the `regular` for the second. We followed this path as well but it turned out that results were not any better than considering only x variables.

Table 4.10 and 4.11 show respectively the results for balanced and non-balanced instances

Table 4.10 Average results for 40 TTPPV balanced instances (model 1)

TTPPV1 - Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	5.3	41.2	71186	0.9	252	97.00%
Brélaz; lexico	0.1	60.1	100661	0.1	0	95.00%
dom/ddeg; lexico	0.1	0.1	7	0.1	0	100.00%
IBS	8.1	8.1	4694	6.0	4320	100.00%
IBS+	12.1	15.0	12513	9.3	8098	99.75%
Ilog IBS+	12.8	12.8	9208	9.3	7949	100.00%
RSC+LA	226.0	226.0	0	134.9	0	100.00%
RSC2+LA	39.3	39.3	0	25.9	0	100.00%
Brélaz; maxSD	0.2	120.2	175869	0.3	0	90.00%
dom/ddeg; maxSD	19.8	108.3	151076	0.2	0	92.50%
IBS; maxSD	8.6	8.6	5317	6.3	4584	100.00%
IBS+; maxSD	8.8	8.8	5317	6.4	4584	100.00%
maxSD	0.4	0.4	6	0.3	0	100.00%
maxAggr(aAvg)	0.4	90.4	133863	0.6	0	92.50%
maxAggr(avgSDDisc)	0.7	60.7	59963	0.5	0	95.00%
maxAvgVar; maxSD	0.4	30.4	39395	0.4	0	97.50%

for the first model proposed. A % solved vs time plot for non-balanced instances is reported in Figure 4.12.

We confirm the results reported in [74] where they proved that balanced instances are relatively easy to solve. Almost all the heuristics are able to solve the whole instance set within few seconds. The only heuristic that falls short here is RSC+LA whose overhead confirms again to be a limit. We remark also that hybrid heuristics, particularly Brélaz; maxSD and dom/ddeg; maxSD, do not perform better than their pure counterpart.

Differences in heuristic performance are better sorted out when analyzing the non-balanced instances. No heuristic is able to solve all of the benchmark set, however impact-based heuristics have an edge over the rest. Hybrid heuristics IBS; maxSD and IBS+; maxSD perform slightly better than IBS and IBS+, whereas the pure maxSD struggles much more (only 15% instances solved); therefore, the winning aspect of hybrid heuristics comes from the variable selection rather than value selection. In general, counting-based heuristics perform very badly and they have a worse performance than a very simple heuristic such as rndMinSizeRndVal.

TTPPV - Model 2

The alternative model (referred to as TTPPV2) is the following:

Table 4.11 Average results for 40 TTPPV non-balanced instances (model 1)

TTPPV1 - Non-Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	20.6	770.2	1464778	455.3	767090	36.25%
Brélaz; lexico	0.2	960.0	2016570	186.1	239716	20.00%
dom/ddeg; lexico	2.3	870.7	1854841	111.5	152393	27.50%
IBS	58.2	514.9	561102	85.5	89597	60.00%
IBS+	49.6	512.6	705301	219.2	304771	59.50%
Ilog IBS+	25.9	554.2	853358	96.0	110422	55.00%
RSC+LA	232.8	643.9	608	340.5	0	57.50%
RSC2+LA	30.9	937.0	145440	472.6	0	22.50%
Brélaz; maxSD	0.2	1050.0	2011554	387.4	475368	12.50%
dom/ddeg; maxSD	0.5	1020.1	1962969	329.0	0	15.00%
IBS; maxSD	150.9	518.1	516610	114.2	110718	65.00%
IBS+; maxSD	154.4	520.4	505082	116.2	109453	65.00%
maxSD	58.5	1028.8	1423450	420.2	0	15.00%
maxAggr(aAvg)	0.4	870.1	1208806	129.1	0	27.50%
maxAggr(avgSDDisc)	15.6	903.9	1076904	177.3	118206	25.00%
maxAvgVar; maxSD	0.8	930.2	1215478	189.3	0	22.50%

$$
\begin{array}{ll}
x_{i,j} = k \iff x_{k,j} = i & 1 \leq i \leq n, 1 \leq j \leq n-1 \\
x_{i,j} \neq i & 1 \leq i \leq n, 1 \leq j \leq n-1 \\
\texttt{alldifferent}((x_{i,j})_{1 \leq j \leq n-1}) & 1 \leq i \leq n \\
\texttt{regular}((x_{i,j})_{1 \leq j \leq n-1}, PV_i) & 1 \leq i \leq n \\
\texttt{alldifferent}((x_{i,j})_{1 \leq i \leq n}) & 1 \leq j \leq n-1 \\
x_{i,j} \in \{1, \ldots, n\} & 1 \leq i \leq n, 1 \leq j \leq n-1,
\end{array}
$$

This model is basically the same as TTPPV1 [7] except that the home-away pattern is defined directly on the x variables. The difference between the two models reside on the available counting information: TTPPV2 provides counting information directly on the branching variables x for all the global constraints. On the other hand, in the TTPPV1 the branching algorithms can rely only on counting `alldifferent` constraints since the `regular` constraints are posted on auxiliary variables not promoted to the status of branching variables. From experimental results, this choice makes a considerable difference in solving the problem instances through counting-based heuristics.

Results of the TTPPV2 model for balanced and non-balanced instances are reported

7. TTPPV1 presents also more auxiliary variables and related channeling constraints that have been omitted to give an high level view of the model

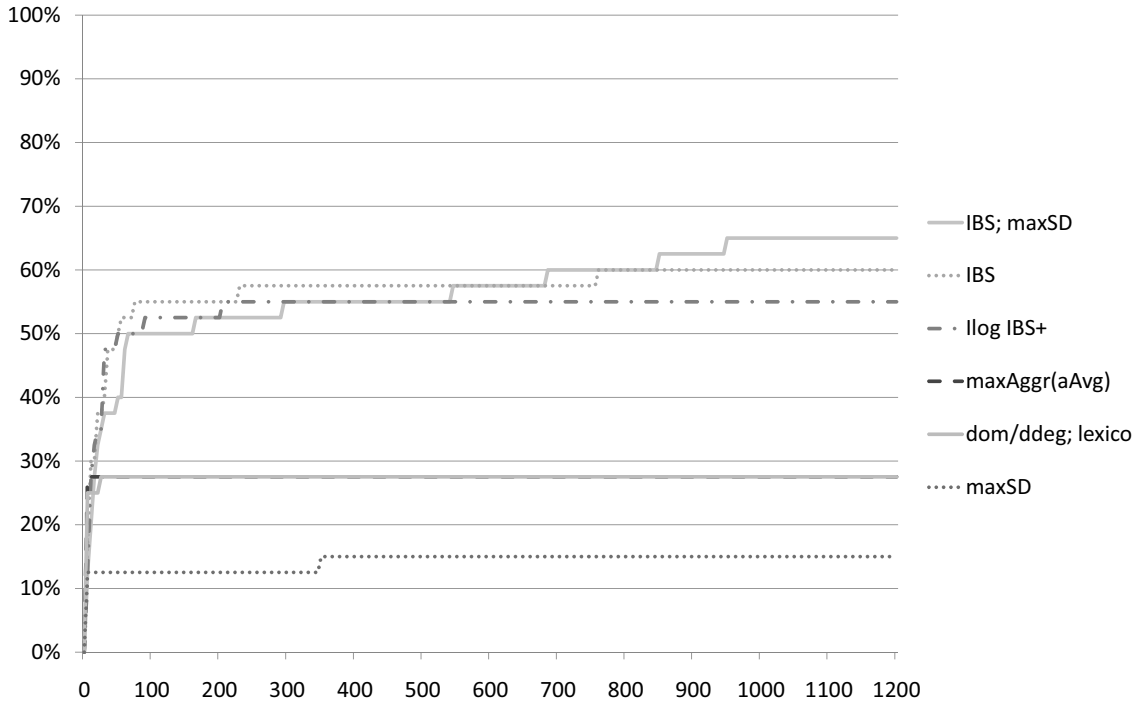


Figure 4.12 Percentage of non-balanced instances solved vs time for TTPPV problem (first model - TTPPV1)

respectively in Table 4.12 and 4.13; for non-balanced instances we report a plot of the percentage of solved instances vs time in Figure 4.13.

The situation remains roughly unchanged w.r.t. TTPPV1 for what concerns the balanced instances. Counting-based heuristics show a slight improvement as they are now able to solve all the instances (in TTPPV1 only maxSD succeeded on all the balanced instances). As in TTPPV1, hybrid heuristic degredate slightly the performance w.r.t. traditional heuristics.

Counting-based heuristics lead the group when it comes to non-balanced instances. maxSD is close to solving all the instances and interestingly maxAggr(avgSDDisc) closes this benchmarks set; the majority of the instances are solved with the same effort as for balanced instances. Five out of six of the instances that are infeasible have been proven so by maxAggr(avgSDDisc) in less than 2 seconds, with only one instance taking a bit more than 4 minutes. Note that this counting-based heuristic was conceived particularly keeping the *first-fail* principle in mind and in this benchmark set it proved in fact that it is well suited also for proving infeasibility. Its total number of backtracks is between 3 and 4 orders of magnitude better than traditional heuristics enforcing the same level of consistency.

Surprisingly, hybrid heuristics gave worse results than traditional ones, therefore proving that, for this problem, the lead of counting-based heuristics comes from the variable selection

Table 4.12 Average results for 40 TTPPV balanced instances (model 2)

TTPPV2 - Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	2.5	11.5	34220	0.2	15	99.25%
Brélaz; lexico	0.1	0.1	9	0.1	0	100.00%
dom/ddeg; lexico	0.1	0.1	27	0.1	0	100.00%
IBS	8.0	8.0	5189	6.0	4488	100.00%
IBS+	10.9	10.9	8287	8.4	7583	100.00%
Ilog IBS+	10.8	10.8	8250	8.4	7555	100.00%
RSC+LA	207.6	207.6	0	131.9	0	100.00%
RSC2+LA	37.7	95.8	18155	32.9	0	95.00%
Brélaz; maxSD	0.1	180.1	436244	0.4	0	85.00%
dom/ddeg; maxSD	8.1	157.1	391736	0.5	0	87.50%
IBS; maxSD	8.7	38.4	22679	7.2	5630	97.50%
IBS+; maxSD	8.6	38.4	22303	7.2	5627	97.50%
maxSD	0.5	0.5	0	0.4	0	100.00%
maxAggr(aAvg)	0.5	0.5	0	0.4	0	100.00%
maxAggr(avgSDDisc)	0.6	0.6	1	0.5	0	100.00%
maxAvgVar; maxSD	0.6	0.6	0	0.4	0	100.00%

heuristic.

Comparing alldifferent counting algorithms in the TTTPV

As for the Latin Square problem, we tested the different counting algorithms for the `alldifferent` constraint presented in Chapter 3. Particularly, we tested the maxSD heuristic with the pure sampling algorithm – maxSD sampl-DC, an exact enumeration followed by a sampling algorithm – maxSD exact+sampl-DC and the upper bound algorithm enforcing three consistency level – maxSD UB-DC, maxSD UB-AC and maxSD UB-FC (see Chapter 3 and Section 4.3.2 for further information). Results are reported in the Table 4.14; Figure 4.14 shows the percentage of solved instances vs time.

The sampling algorithm shows its main drawbacks i.e. it is not competitive in solving easy instances: the number of backtracks is low indeed but the time spent in sampling is simply a waste of time in easy problems but crucial in difficult ones. Exact enumeration adds another consistent overhead to the counting procedure with the results of being three orders of magnitude slower than upper bounds based on arc consistency or forward checking. These latter are about 10 times faster than upper bounds based on domain consistency.

Non-balanced instances are harder to solve and none of the heuristics was able to solve all

Table 4.13 Average results for 40 TTPPV non-balanced instances (model 2)

TTPPV2 - Non-Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	15.7	768.8	2352684	376.2	952225	36.25%
Brélaz; lexico	4.1	901.0	2753370	128.6	164810	25.00%
dom/ddeg; lexico	4.7	901.2	2773954	135.6	195813	25.00%
IBS	67.1	775.2	1209574	209.0	271429	37.50%
IBS+	62.7	515.2	921688	307.5	561356	59.25%
Ilog IBS+	64.0	632.0	1056281	108.8	139389	50.00%
RSC+LA	205.5	553.6	697	284.5	0	65.00%
RSC2+LA	43.8	911.0	203565	486.5	0	25.00%
Brélaz; maxSD	0.1	1170.0	2671953	936.8	1854798	2.50%
dom/ddeg; maxSD	0.1	1170.0	2685464	932.6	1870735	2.50%
IBS; maxSD	209.4	952.4	916441	642.6	613131	25.00%
IBS+; maxSD	207.5	951.9	911345	642.7	611726	25.00%
maxSD	0.6	30.5	2855	0.5	0	97.50%
maxAggr(aAvg)	0.8	30.8	2914	0.5	0	97.50%
maxAggr(avgSDDisc)	7.5	7.5	682	0.5	0	100.00%
maxAvgVar; maxSD	0.6	60.6	16937	0.6	0	95.00%

the 40 instances. Counting based on upper bounds allowed to cut computing time by almost 80% w.r.t. the sampling algorithm and by 85% w.r.t. exact enumeration and sampling. The number of backtracks of maxSD UB-X reflects what was expected in Chapter 3: the stronger is the consistency level the more informative is the counting information. maxSD UB-DC has however a more significant overhead when compared to maxSD UB-AC and maxSD UB-FC. The last two have very similar performance. Figure 4.14 shows how counting based on upper bounds has a clear edge when compared to the other methods.

4.4 Summary and conclusions

We proposed in this section a variety of heuristics based on counting information. We tested them on eight different problems and getting rather convicing results. We report in Table 4.15 aggregated numbers showing arithmetic and geometric averages of the average results obtained on the eight problems (for TTPPV we considered the model TTPPV2).

Hybrid heuristics are not as efficient as counting-based heuristics, nonetheless for what concerns IBS; maxSD and IBS+; maxSD they improve over pure impact-based heuristics; dom/ddeg and Brélaz do not improve nor degrade their performance when hybridized. Counting-based heuristics are indeed the one solving the highest percentage of instances, with the lowest

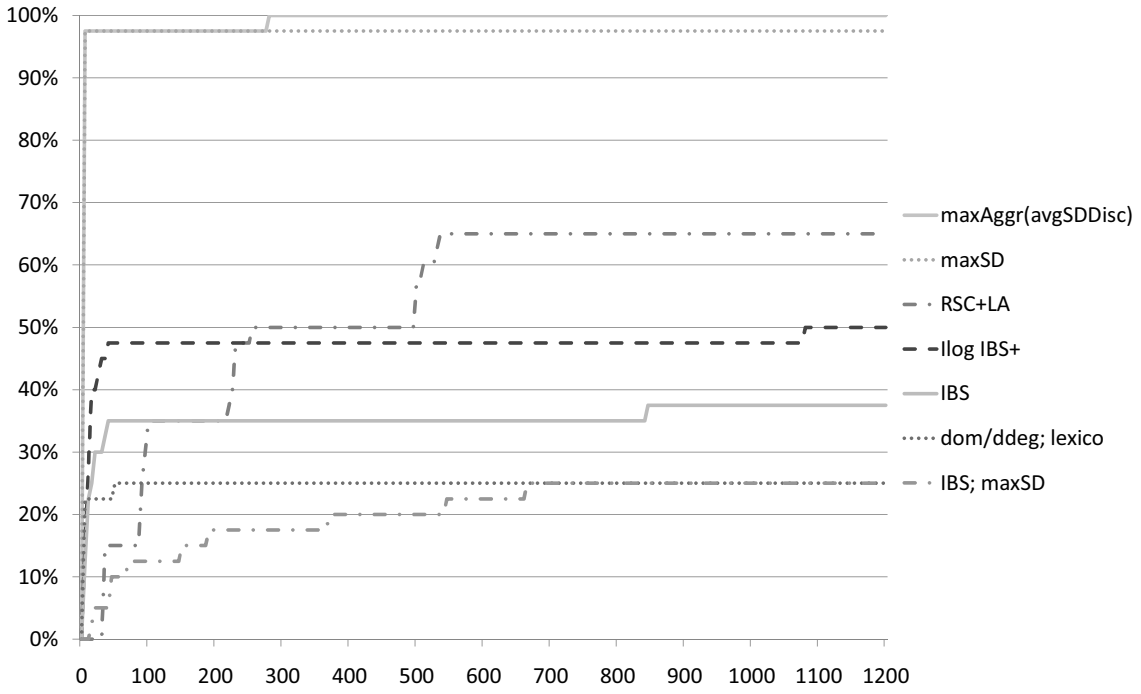


Figure 4.13 Percentage of non-balanced instances solved vs time for TTPPV problem (second model - TTPPV2)

running time and with the lowest number of backtracks when compared to heuristics enforcing the same level of consistency. Despite not being the best heuristic for all the problems tested, they can be considered the best overall. Nonetheless, they present some drawbacks and raise some questions that need to be addressed in future research. We will describe next the pitfalls and some points for discussion.

Local view vs global view Counting-based heuristics have local yet very precise information based on individual constraints. Impact-based heuristics (both IBS and RSC+LA) use a different approach by trying to infer the size of the search tree to be explored; they have a more global view on the problem model, however the information on which they rely is coarser and not as precise as counting. We believe that in the benchmarks in which counting-based heuristics were worse than impact-based heuristics, this was mainly due to the lack of a global view as in impact-based search. The aggregation functions we proposed partially overcome this limit, however future research should focus on some kind of more complex and effective aggregation. An interesting approach has been already proposed in [15] and possibly a tighter integration of machine learning and counting could be a possible answer for the trade-off between global coarse information vs local fine grained information. We would like

Table 4.14 Average solving time (in seconds) and number of backtracks for 40+40 balanced and non balanced instances of the TTPPV.

	balanced			non-balanced		
heuristic	a.T	a.Bcks	%sol	a.T	a.Bcks	%sol
maxSD sampl-DC	25.9	2	100%	140.7	3577	91%
maxSD exact+sampl-DC	120.4	1	100%	216.9	1210	91%
maxSD UB-DC	6.7	1	100%	36.8	245	98%
maxSD UB-AC	0.6	1	100%	30.6	2733	98%
maxSD UB-FC	0.5	1	100%	30.5	2906	98%

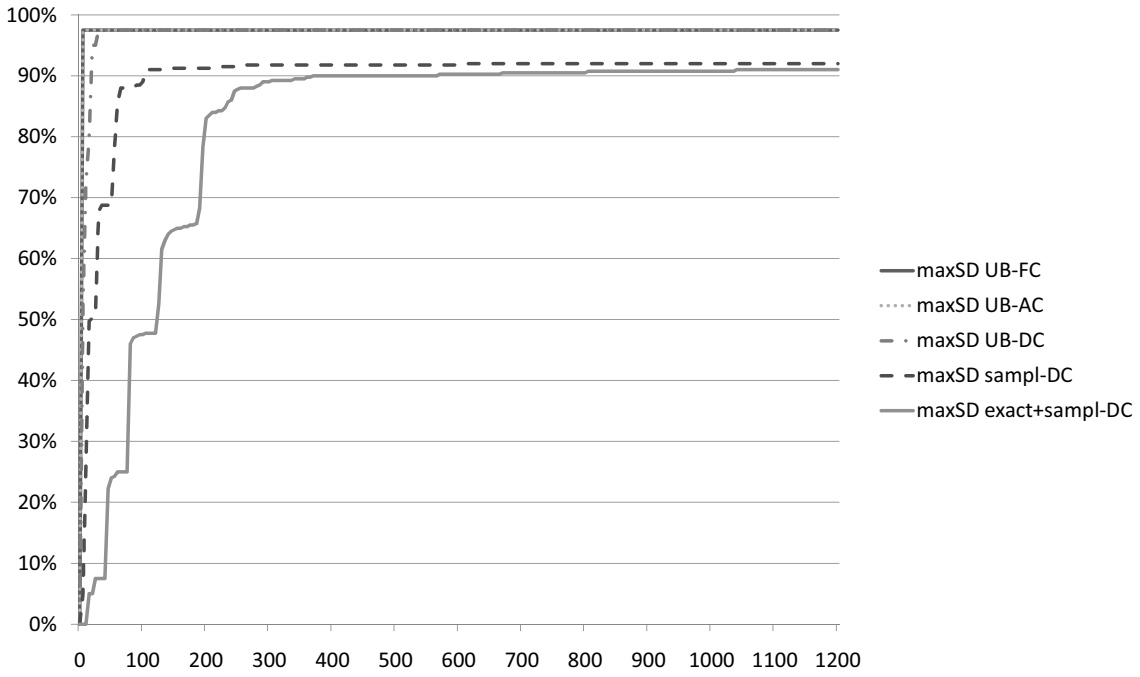


Figure 4.14 Comparison of maxSD heuristic with different counting algorithms: percentage of solved instances vs time in seconds for non balanced instances of the TTPPV.

also to mention an unsuccessful attempt: we designed and implemented heuristics inspired by impacts but that still consider solution count. We defined the solution count impact as the reduction of the Cartesian product of the constraints' solution count (or tightness of the constraints). More formally:

$$I(x_i = a) = 1 - \frac{P^*_{after}}{P^*_{before}}$$

where P^* can be either $P^{SC} = \prod_\gamma \#\gamma$ or $P^T = \prod_\gamma T_\gamma$ after and before the instantiation of $x_i = a$. The heuristic resembles closely impact-based search but considering instead the

Table 4.15 Aggregated average results over the eight problem domains

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	123.2	408.7	708994	347.9	241614	71.80%
Brélaz; lexico	94.0	447.0	690109	366.7	286357	67.55%
dom/ddeg; lexico	102.0	453.8	687008	370.7	290280	67.55%
IBS	70.5	352.7	237088	120.7	38937	74.90%
Ilog IBS+	197.0	367.9	315401	132.7	63062	82.59%
IBS+	252.4	423.6	422921	145.7	82068	77.70%
RSC+LA	313.5	536.2	4033	196.2	141	69.38%
RSC2+LA	187.9	536.0	49248	276.7	4788	64.48%
Brélaz; maxSD	104.1	458.6	1226556	337.3	230290	67.95%
dom/ddeg; maxSD	103.4	454.5	1235699	335.6	231490	68.21%
IBS; maxSD	132.9	307.8	162754	114.5	28293	84.27%
IBS+; maxSD	134.6	309.5	162263	115.1	26219	84.27%
maxSD	48.6	84.8	75450	31.7	5390	96.95%
maxAggr(maxRelSD)	57.8	115.9	48810	18.8	1168	94.86%
maxAggr(aAvg)	64.6	154.3	236063	81.1	20618	92.24%
maxAggr(wAntiTAvg)	49.5	94.3	82113	31.1	9467	96.16%
maxAggr(avgSDDisc)	89.5	189.3	95446	36.4	3300	89.81%

reduction of the solution count or of the tightness.

Unfortunately, these heuristics and some variations of them were not able to compete with the maxSD heuristic.

Counting algorithms The Global Constraint Catalog [9] counts now a few hundred constraints of which a few dozen have been implemented in solvers; counting algorithms have been proposed only for very few of them and none has been implemented in any solver yet. This is actually a limit for a broad applicability of counting-based heuristics. We believe that generic sampling techniques for counting the number of solutions of any kind of constraint are not applicable as the overhead might be too high. We implemented a preliminary version of a generic sampler inspired by the one implemented for the `alldifferent` constraint; results have been poor due to the inherent overhead, similar to the one experienced for the `alldifferent` constraint. Therefore, as it has been for constraint filtering algorithms, a lot of research effort must be put into designing counting algorithms if counting-based heuristics need to be systematically applied. Fortunately, counting is a field of mathematics and computer science that has been largely studied, therefore ideas and solutions might be directly borrowed from previous works.

Model views Users often need to introduce auxiliary variables or different views of the models that are linked together by channeling constraints. However, we have seen for the TTPPV how important it is to provide all the counting information available at the level of the branching variables or at least at some level where direct comparison of solution densities is meaningful. Channeling constraints that are relation one to many (such as the one present in the TTPPV) can be dealt with parameterized counting algorithms. As an example, a "home-away" schedule for a team might still be bound to different opponents, that is, an individual solution of the constraint defined over the home-away variables can still represent many solutions for the branching variables. Parameterized counting algorithms would introduce a factor for each individual solution to be considered when computing the solution densities; this can be easily done for the `regular` constraint by associating a factor to each of the arcs present in the layered graph: one path from source to sink would therefore count a number of solution equivalent to the product of the factors. Similar reasoning could be done for the `alldifferent` constraints where $0-1$ entries in the adjacency matrix for the computation of the permanent would be replaced by more general $0-n$ entries.

Multiple views on the model linked through more complex channeling constraints represent however a limitation in the current framework.

Optimization problems Optimization problems have not been touched in this chapter. Heuristics with a strong emphasis on feasibility (such as counting-based heuristics) might not be well suited for problems with a strong optimization component, yet very useful when dealing with optimization problems that involve hard combinatorics. Ideally, counting algorithms should not be blind to cost reasoning and in this respect we mention another interesting preliminary theoretical results we got that still needs to find an immediate applicability; for the `regular` constraint we designed and implemented an algorithm that not only counts the number of solutions that involve a particular variable-value pair but that can also return the average cost of all the solutions employing that particular variable-value pair. The basic idea is similar to the counting algorithm, i.e., for each node of the layered graph we compute the sum of the costs of all the paths reaching that node and the sum of the costs of all the paths leaving that node. We think that this kind of information might be practically useful when solving optimization problems.

To conclude, counting-based heuristics have proven to be very efficient and effective; counting algorithms can be employed as efficient building blocks for designing particularly informed heuristics. Nonetheless for a completely generic and automated search heuristic the points described above must be addressed.

CHAPTER 5

Experimental Analysis of Quick Shaving

Strong forms of consistency have proven to be a successful element in tackling hard combinatorial problems with Constraint Programming [1]. The focus has moved recently from local consistencies to more powerful singleton consistencies. Singleton consistency ensures that a variable-value assignment does not lead to an immediate failure after the constraint network has been propagated. Although singleton consistencies undoubtly reduce the search space, it is still not unequivocally accepted whether this reduction outweighs the computational overhead. What makes singleton consistency particularly heavy is the temporary assignment (probe) of a variable to a value and the following propagation over the whole constraint network just to verify if it is singleton consistent. Furthermore this additional computational effort is only rewarded by some search space reduction when the temporary assignment leads to failure.

Singleton consistency has been applied originally to scheduling [72] (under the name of shaving) and to continuous CSPs [66] and was later on formalized by Bessière et al. in [24]. Their solution (Singleton Arc Consistency – SAC) enforces singleton consistency at every node of the search tree on each variable-value pair while keeping the underlying constraint arc consistent; the procedure is iterated until no more pairs can be filtered. Prosser et al. in [82] extended this idea and proposed singleton global arc consistency (SGAC) in which they enforce global arc consistency on the constraint network. In order to overcome the significant overhead they also proposed a reduced form of singleton consistency in which, at a given node, each variable-value pair is checked only once. Barták et al. improved SAC in [5] by introducing some sort of support bookkeeping that avoids unnecessary work; the worst-case time complexity remains the same as SAC but experimental results show that their solution outperforms the original SAC. Although Bessière et al. in [11] proposed an algorithm (SAC-Opt) with an optimal time complexity, singleton consistency techniques cannot be blindly applied to any problem. In this respect, Stamatatos et al. in [100] proposed a random probing preprocessing procedure that tries to learn the consistency level to apply to each constraint; they show promising results on some problems with binary constraints, however the effectiveness of the algorithm needs to be verified when applied to models containing global constraints.

In order to further lighten the overhead of singleton consistency, some other approaches

1. Parts of this chapter appeared in [115]

with a reduced inference power have been developed. Particularly, Lhomme proposed in [67] a reduced form of singleton consistency called Quick Shaving (QS) that, instead of working proactively at each node of the search tree, verifies reactively only variable-value pairs that have recently caused a failure.

In [101], Szymanek et al. proposed to integrate QS with a procedure that extract, from the problem constraints, variable-value pairs that may be good candidates for shaving. Their method shows an advantage over pure QS however, up to now, they provide algorithms to advise shavable variable-value pairs only for the `all-different` and `sum` constraints.

Pure QS has a theoretically proven small overhead but still not all problems can benefit from applying such a technique. Ideally all the added probing overhead of QS should be rewarded by a significant pruning of the search space. This depends mainly on two factors: firstly how often the probing detects that a value should be filtered, secondly the impact of the propagation induced by filtering a value through QS (i.e. how much the search space is reduced after shaving a value). To date there is no clear picture of whether or not QS can improve the solving time on a given problem class or instance; furthermore, this possible performance gain also depends on the search heuristics employed during the search i.e. there are problem classes for which some heuristic can improve with QS enabled and some other not.

This study tries to shed some light on these issues. Particularly, we propose a first coarse grain feature that could help in the choice of enabling QS during search. This study is not intended to give a definitive answer but rather it is a first step in the direction of understanding the behavior of QS.

The rest of this chapter is organized as follows. Section 5.1 reviews the quick shaving algorithm. Section 5.2 is dedicated to an experimental analysis of QS. In Section 5.3 we propose a simple method that improves the behavior of QS. Concluding remarks are then made.

5.1 Quick Shaving

We briefly introduce some concepts that will be used throughout the chapter.

Definition 27. *Given a problem P, a value j of a variable x_i is said to be* shavable *if θ-consistency*[2] *is able to detect inconsistency on problem $P' = P \cup \{x_i = j\}$. We call* shaving attempt *a probe on an assignment $x_i = j$ to test whether it is shavable or not.*

Definition 28. *The* shaving ratio *is defined as the number of shaving attempts that actually*

2. any form of consistency

prove a value to be shavable for a certain variable over the total number of shaving attempts on that variable-value pair.

The basic idea of quick shaving [67] is that each variable-value pair that is detected to be shavable in a branch at depth $k+1$ should be checked for shaving at depth k. How can we detect values that are shavable at level $k+1$? From failures, that is if an assignment leads to a failure (a domain wipeout) then the same assignment should be checked for shaving higher up in the search tree. Algorithm 10 shows the pseudo-code of a modified search procedure that integrates QS. The input parameters of the procedure are the current problem P and a branching decision ct; *shavableList* is an output parameter. Initially the algorithm acknowledges the branching decision (lines 1-2): if a failure is detected then the search backtracks and we annotate the decision that leads to failure (line 4). Otherwise we proceed with a binary search tree trying to assign the unbounded variables: the algorithm selects the next branching point and creates an OR node with the decision and its negation (respectively lines 8 and 13). If the left branch succeeds then a solution has been found and the search returns true; if it fails then on the backtrack we trigger the shaving procedure - line 12. The shaving procedure (see Algorithm 11) probes all the variable-value pairs collected in the left branch and it filters those that are actually shavable (those that are not are removed from the shaving list). Note that in case a branching decision and its negation are shavable then the Shave algorithm returns right away a failure. In such a case, Algorithm 10 backtracks and propagates up in the search tree the shaving list (lines 19 to 21). Otherwise, it creates the right branch (line 13); in case the right branch does not succeed the shaving information collected from the left and right branches are propagated up in the search tree.

This is the basic algorithm but, as shown in Example 10, some minor optimizations are possible.

Example 10. *Figure 5.1 shows an example of quick shaving. Nodes are labeled in order of their exploration. The first node that causes a failure is node 3 and the shaving attempt $x_i = j$ is propagated up to node 2 for an eventual shaving. Here we see the first optimization: it is superfluous to probe for shaving $x_i = j$, since the search is about to branch on $x_i \neq j$. The pair $x_i = j$ will be kept in the shaving list and propagated up to node 1 for a shaving attempt.*

The right child of node 2 is then created. Both children of node 4 (nodes 5 and 6) bring a failure, therefore node 4 ends up with a shaving list $SL_4 = \{(x_{i'} = j'), (x_{i'} \neq j')\}$. This shaving list will then be propagated up to node 2 whose final shaving list will be $SL_2 = \{(x_i = j), (x_{i'} = j'), (x_{i'} \neq j')\}$. After backtracking, at node 1, shaving probing is finally triggered, attempting to shave the elements of SL_2. Those that are shavable are kept in the shaving list

```
1  add ct to P;
2  enforce θ-consistency;
3  if failure then
4      shavableList ← {ct};
5      return false;
6  if some variables are unassigned then
7      dec ← select a branching decision;
8      retLeft ← QS-Search(P,dec, shavableLeft);
9      if retLeft = true then
10         return true;
11     else
12         if Shave(P,shavableLeft) then
13             retRight ← QS-Search(P,not(dec), shavableRight);
14             if retRight = true then
15                 return true;
16             else
17                 shavableList ← shavableLeft ∪ shavableRight;
18                 return false;
19         else
20             shavableList ← shavableLeft;
21             return false;
22 else
23     return true;
```

Algorithm 10: QS-Search(in P, in ct, out shavableList)

```
1  foreach dec ∈ shavableList do
2      ret ← θ-consistent(P ∪ dec);
3      if ret = failure then
4          P ← P ∪ not(dec);
5          retNeg ← θ-consistent(P);
6          if retNeg = failure then
7              return false;
8      else
9          shavableList ← shavableList \ {dec} ;
```

Algorithm 11: Shave(inout P, inout shavableList)

of node 1 and possibly passed further up in the search tree in case of backtrack; the ones that are not shavable are discarded.

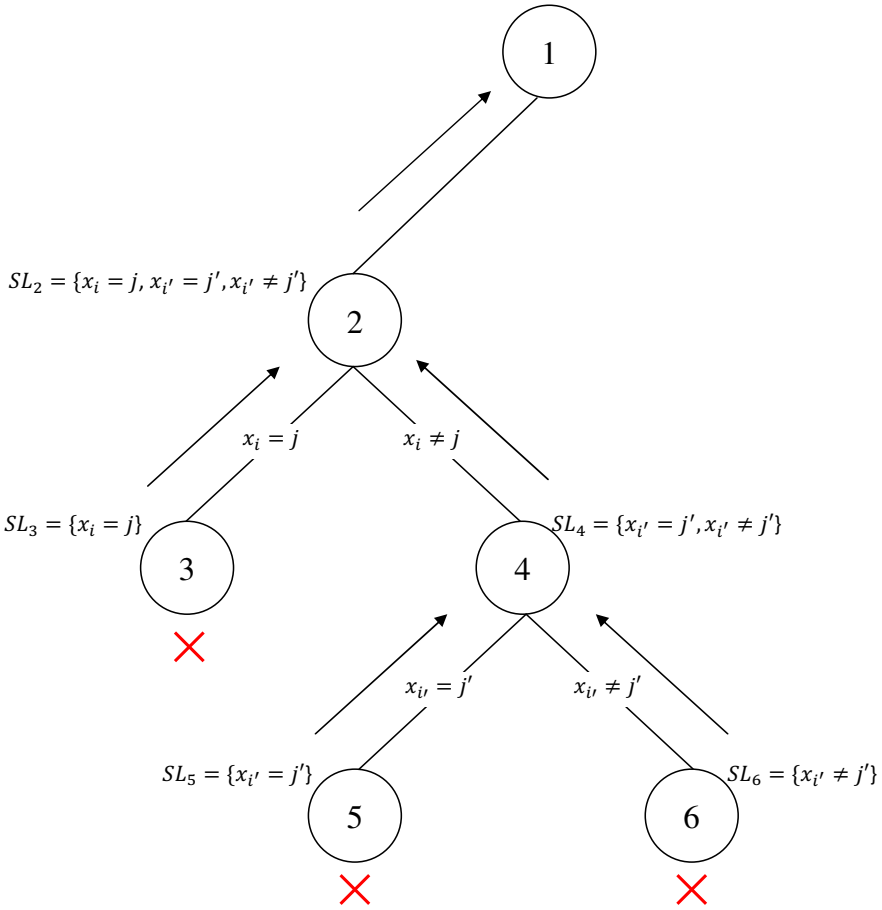


Figure 5.1 Quick Shaving Example

5.2 Understanding Quick Shaving Behaviour

There is undoubtedly a computational price to pay for QS. So in what circumstances should it be applied? We conjecture that a high shaving ratio leads to an actual computational advantage in the search. To put this conjecture to the test, we performed an empirical study over nine different problem types (for a total of about 630 instances) using six different heuristics with and without the quick shaving procedure (for a total of about 640 hours of testing). With one exception, all the heuristics over all the problems showed a very similar behavior: when the shaving ratio is higher than a certain threshold then the total solving time is lower compared to the same heuristic without quick shaving. This study has been conducted with two main aims: firstly, to collect empirical evidence supporting the conjecture; secondly, to determine whether this shaving ratio threshold is an invariant with respect to problem class, instance, and search heuristic.

5.2.1 Problems

We tested with the problems presented in Chapter 4 (for the TTPPV, we used the model 1) and one additional described next.

Random CSP We generated 245 instances of random CSP following a random model defined by four parameters $< n, d, p_1, p_2 >$ (n variables with domain size of d, choosing $\frac{p_1 n(n-1)}{2}$ binary constraints and $p_2 * d^2$ forbidden tuple in each constraint). We used a generator [3] that implements a Model B generator (see [36]: n has been set to 150, d to 7 and we varied p_2 from 0.2 to 0.8 with steps of 0.1 and κ from 0.8 to 1.1 with steps of 0.05 (phase transition i.e. hard instances occurs around $\kappa \approx 1$). For each combination of p_2 and κ we generated 5 instances with p_1 computed accordingly to the formula (see [35]):

$$\kappa = \frac{-|C| \log_2(1 - p_2)}{n \log_2(d)}$$

where C is the set of binary constraint.

5.2.2 Heuristics

The six heuristics we tested are briefly described here (see Chapter 4 for further information on the heuristics):

- Brélaz; lexico ([17]): it selects the set of variables with the smallest domain and then it breaks ties choosing the variable with highest dynamic degree. Value selection is performed in lexicographic order.
- dom/ddeg; lexico ([13] and [97]): it selects the variable that minimizes the domain cardinality over the dynamic degree. Value selection is performed in lexicographic order.
- Impact Based Search (IBS) [86]: it selects first the variable whose instantantion triggers the largest search space reduction (highest impact) that is approximated as the reduction of the Cartesian product of the variables' domains. Then it selects the value with the smallest impact.
- Maximum Solution Density (maxSD): it extracts solution counting information from the constraints and then it branches on the variable-value pair with the highest solution density (see Chapter 4). We use the counting algorithms presented in Chapter 3 for the `regular`, `pattern` constraint (special case of `regular`), and for the `alldifferent` constraint (based on upper bounds) and in [79] for the `knapsack` constraint.

3. http://www.lirmm.fr/~bessiere/generator.html

– Brélaz; maxSD: it selects the variable accordingly with the Brélaz heuristic and then it chooses the value with the maximum solution density.
– Dom/ddeg; maxSD: it selects the variable accordingly with the Dom/ddeg and then it chooses the value with the maximum solution density.

Heuristics based on solution counting assume the presence of some countable constraints in the problem model, therefore the last three heuristics have not been tested on random CSP problems.

5.2.3 Experimental Analysis

Each heuristic has been tested with and without Quick Shaving on all the problem instances described in the previous section (with the exception of random CSPs that have been tested only with Brélaz; lexico, dom/ddeg; lexico and IBS). Every problem has been modelled such that domain consistency is enforced on each constraint. The experiments were run on a AMD Opteron 2.4 GHz and 1GB of RAM with Ilog Solver 6.6. We set a timeout of 20 minutes for each problem instance. As an indication of the problems' difficulty, we present in Table 5.1 the raw results of the heuristics without QS.

To study the performance gain brought by the quick shaving procedure, we define for each heuristic and problem class, the time ratio as the average solving time with QS enabled over the average solving time with QS disabled; a time ratio lower than one indicates on average a performance gain of the heuristic with QS enabled, whereas a time ratio higher than one means a higher solving time with QS. We plot the time ratios in Figure 5.2.

The Latin Square, Nonograms, Rostering and TTPPV are the problems that benefit the most from QS independently from the heuristic (except for IBS in Rostering). The rest of the problems present a slight overhead that becomes important for the Market Split problem and for the MagicSquare. Performance improvements depend mainly on the problem class, nonetheless they may also vary based on the heuristics employed: this is the case, for example, of Nonograms problems in which every heuristic took advantage of QS except IBS; the same happened for Rostering where IBS does better without QS whereas all the other heuristics gained from it. Note also that IBS was the only heuristic for which QS brought advantages just in the Latin Square problem.

Figure 5.3 shows the average shaving ratio for each problem class and heuristic. Problems and heuristics that suffered from QS in terms of solving time are those for which the shaving ratio is low whereas whenever the shaving ratio is sufficiently high we actually have a significant gain. For example in TTPPV, counting based heuristics (maxSD, brelaz;maxSD, domddeg;maxSD) are the ones with higher shaving ratio and this is directly reflected in the time ratio. Note also that the shaving ratio of IBS is in general very low compared to the

Table 5.1 Arithmetic average solving time (in seconds), number of backtracks and percentage of solved instances without quick shaving

	LatinSquare			Nonograms			MultiKnapsack		
heuristic	a.T	a.B	%sol	a.T	a.B	%sol	a.T	a.B	%sol
Brélaz;lexico	751	1458K	48%	297	194K	79%	498	11K	64%
dom/ddeg;lexico	724	1387K	55%	296	195K	79%	500	11K	64%
IBS	716	900K	45%	7	5K	99%	354	4K	76%
maxSD	105	105K	98%	49	18K	97%	73	1K	96%
Brélaz;maxSD	322	569K	83%	295	184K	78%	504	10K	64%
dom/ddeg;maxSD	295	526K	85%	295	187K	78%	501	10K	64%
	MagicSquare			MarketSplit			KPRostering		
	a.T	a.B	%sol	a.T	a.B	%sol	a.T	a.B	%sol
Brélaz;lexico	659	72K	48%	122	110K	100%	679	996K	50%
dom/ddeg;lexico	736	83K	40%	122	110K	100%	681	1004K	50%
IBS	639	24K	50%	202	91K	100%	511	264K	60%
maxSD	12	1K	100%	257	60K	100%	0	0K	100%
Brélaz;maxSD	30	11K	100%	246	93K	100%	655	743K	50%
dom/ddeg;maxSD	138	54K	95%	232	93K	100%	656	749K	50%
	Rostering			TTPPV			RandomCSP		
	a.T	a.B	%sol	a.T	a.B	%sol	a.T	a.B	%sol
Brélaz;lexico	894	8694K	27%	600	972K	50%	199	49K	86%
dom/ddeg;lexico	895	8760K	27%	480	799K	60%	164	38K	89%
IBS	61	76K	97%	373	309K	70%	161	40K	90%
maxSD	935	3133K	23%	540	551K	55%			
Brélaz;maxSD	1122	6984K	7%	780	1113K	35%			
dom/ddeg;maxSD	1122	6969K	7%	697	1005K	45%			

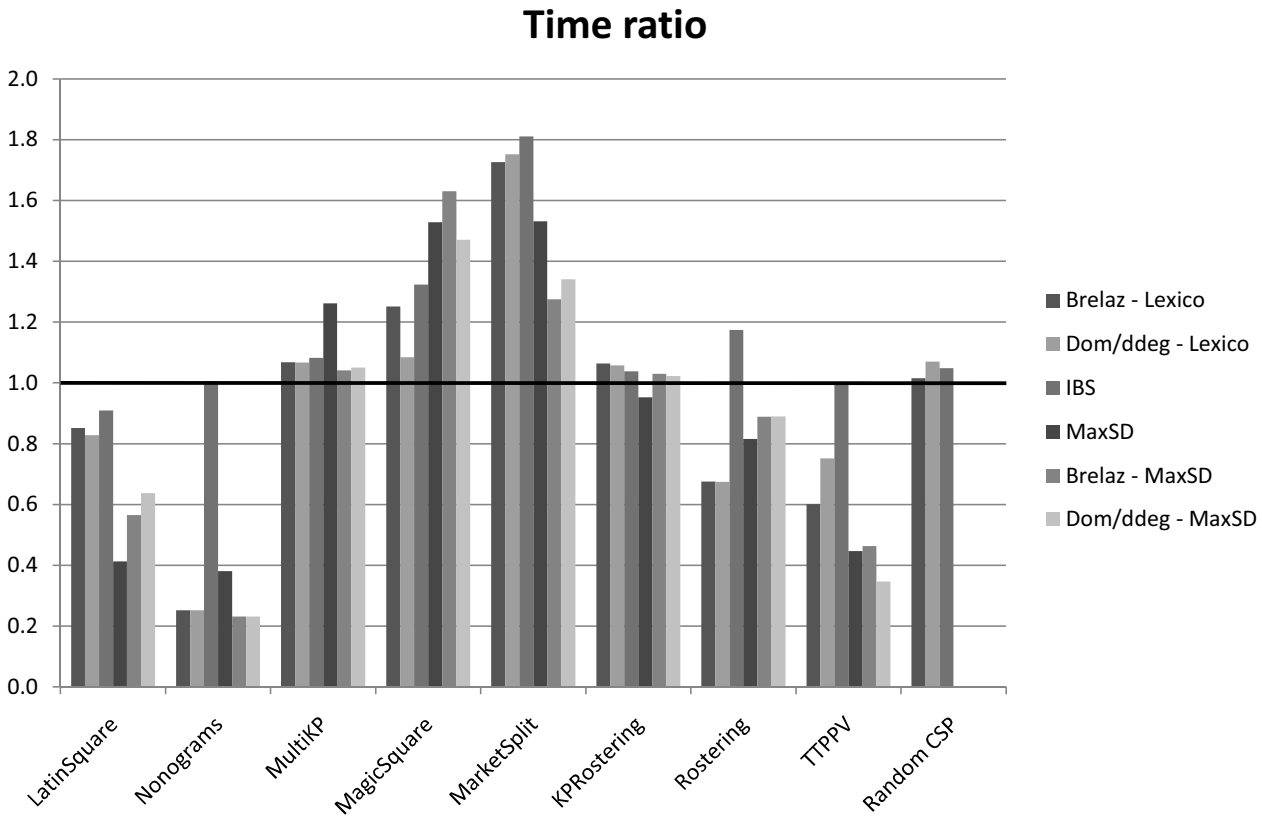


Figure 5.2 Time Ratio: average time with QS over average time without QS

other heuristics.

What sets IBS apart from the other heuristics in this study is possibly its heavy bias toward the fail-first principle. Its low shaving ratio could be explained by the fact that an inconsistency between two variable assignments is detected early on and thus is not much propagated up the search tree through the shaving lists. This leads to a very low shaving ratio. On the other side heuristics such as maxSD lean more toward succeed-first hence, although it is very competitive, an inconsistency may occur between two variable assignments that are relatively far apart on a branch of the search tree. This means that a variable-value pair that causes a fail can be propagated through the shaving list higher in the search tree hence allowing several times the shaving of that pair.

For all the heuristics, we plotted the time ratio vs the shaving ratio of each individual problem instance (630 in total). From the plot, all the instances that were either too easy or too hard (solved within 2 seconds by the heuristic without QS or unsolved by the heuristic with and without QS) were removed, as well as the instances for which the amount of shaving was too low to be statistically significant i.e. with less than 50 shaving attempts; note, as a

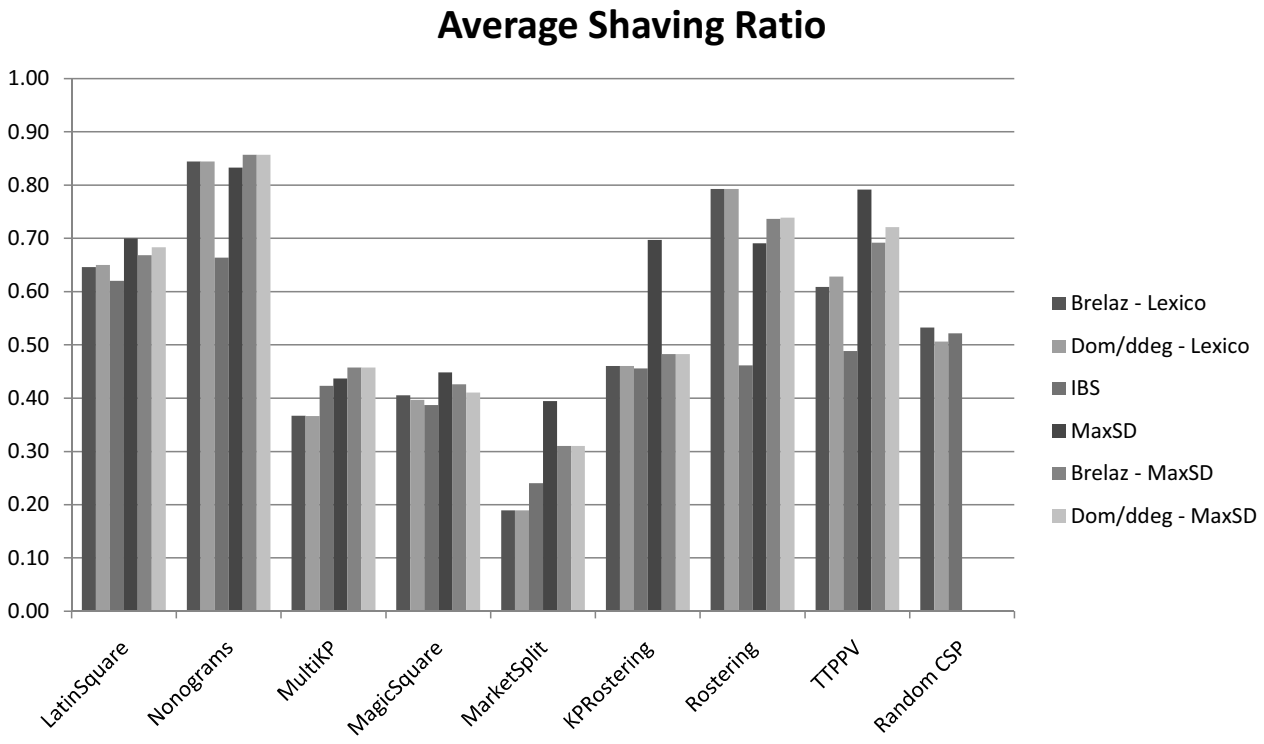


Figure 5.3 Average Shaving Ratio

reference, that the average number of shaving attempts for the Latin Square problem is of the order of 10^5.

Surprisingly, all the plots show a similar behavior: most of the points with a time ratio lower than 1 (performance gain with QS) are situated in an area with a shaving ratio higher than ≈ 0.6. If the shaving ratio goes below this threshold then the time ratio is higher than 1 i.e. QS brings no advantage or even worse it slows down the resolution process. IBS is the only heuristic for which such a split is not as sharp as in the other plots. Figure 5.5 shows how the instances are clustered by problem class for the dom/ddeg heuristic (note that the other heuristics present a similar clustered pattern): for example, nonograms have a time ratio close to 0 with a corresponding very high shaving ratio; Market Split are situated in the area with the lowest shaving ratio. Although different heuristics may skew a bit the plots and have different results (as shown in Figure 5.2), the main factor that influences the shaving ratio is the problem class.

To better investigate the relationship between shaving ratio and performance gain, we present in the following the correlation coefficient between the two. We use the phi-correlation coefficient $\phi = \sqrt{\frac{\chi^2}{N}}$ (where χ is the Pearson's chi-square test and N is the population size) to numerically quantify the correlation shown in the plots (see [19]). The coefficient varies

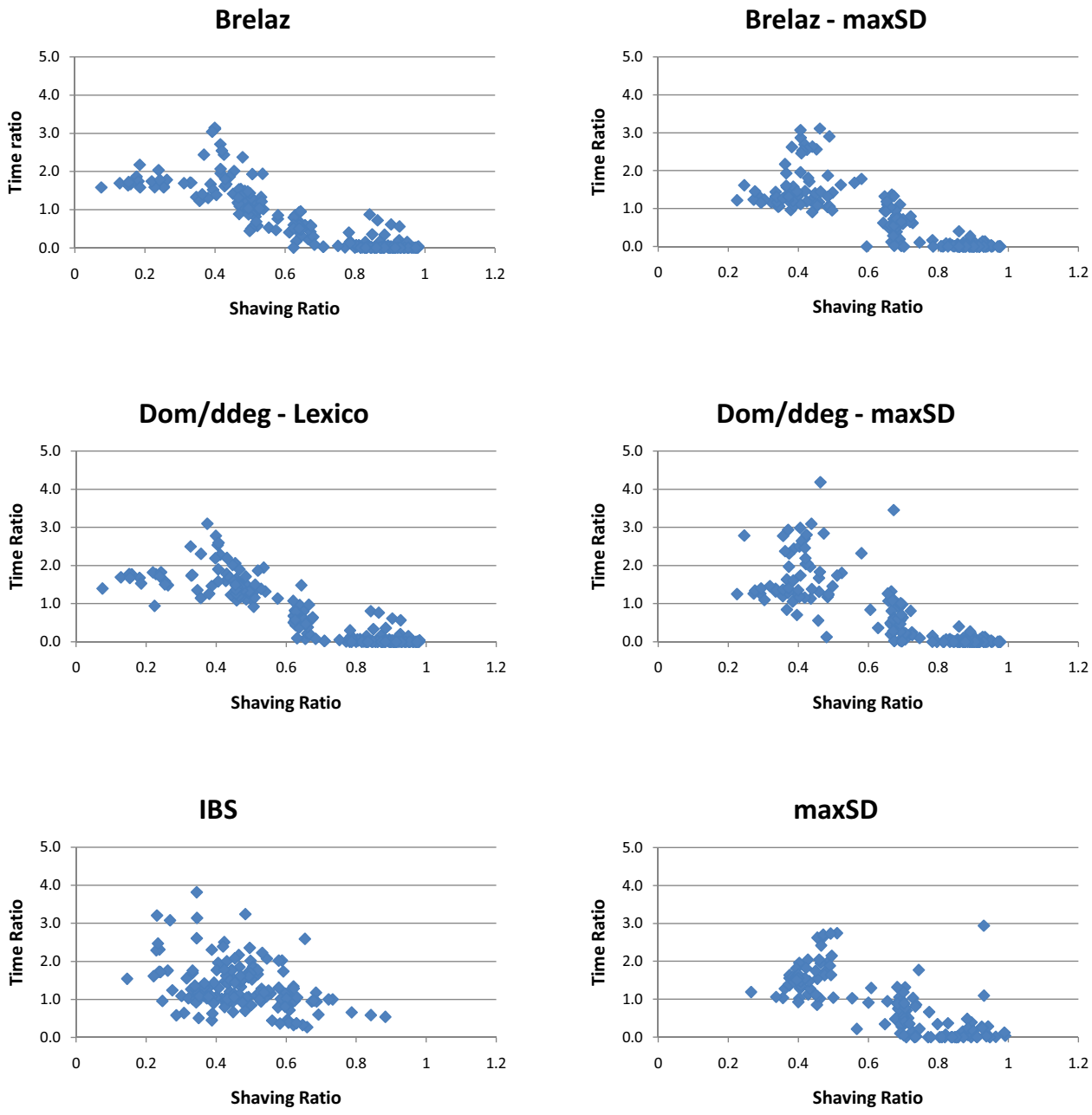


Figure 5.4 Scattered plots

from -1 to 1 where the extremes mean a perfect anticorrelation and correlation and 0 stands for no correlation. This coefficient assumes that the sample population has two features with two values each. In our context the features are the shaving ratio (higher or lower than a given threshold th) and the time ratio (higher or lower than 1). We found the best correlation with $th \in [55\%, 60\%]$. The correlation for different thresholds are shown in Table 5.2. Note that this threshold might be slightly dependent on the shaving implementation, however the

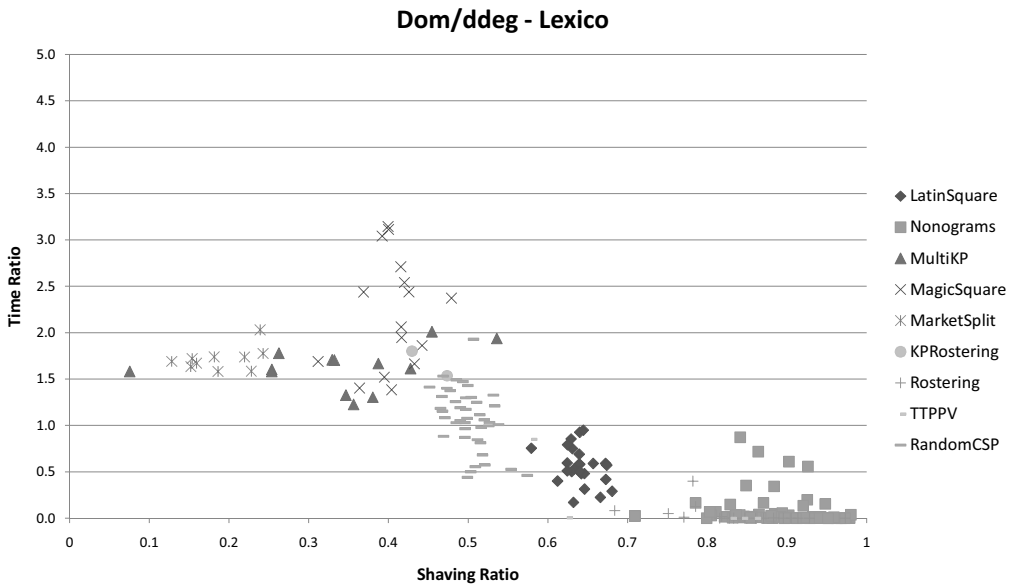


Figure 5.5 Scattered plots

Table 5.2 ϕ correlation coefficient

Heuristic	$th = 50.0\%$	$th = 52.5\%$	$th = 55.0\%$	$th = 57.5\%$	$th = 60.0\%$
Brélaz; lexico	0.85	0.83	0.88	0.86	0.85
dom/ddeg; lexico	0.88	0.92	0.95	0.95	0.96
IBS	0.34	0.44	0.49	0.48	0.49
maxSD	0.81	0.82	0.82	0.82	0.82
Brélaz; maxSD	0.84	0.85	0.85	0.86	0.86
dom/ddeg; maxSD	0.81	0.84	0.84	0.84	0.85

interesting fact is that no matter which implementation you use there is a sharp threshold above which there is an actual gain in employing QS. For sake of clarity this threshold is valid across the heuristics (with some reservation for IBS) and across the problem classes. All the heuristics show a strong correlation while IBS has just a moderate correlation.

5.3 Dynamically Disabling Quick Shaving

Ideally a shaving procedure should justify its probing overhead through a significant reduction of the search space that translates to a computational performance gain. As pointed out earlier, this is not the case for every problem. We can however exploit the behavior of QS — a sharp threshold around a shaving ratio of $55\% \sim 60\%$ — to dynamically disable shaving whenever this threshold is not reached. The procedure keeps track of the

shaving statistics and after K shaving attempts it starts checking the value of the shaving ratio: if it is below the threshold then QS is switched off and never enabled again. For the experimentation, after some pilot experiments, we set K equal to 100, which is a considerably low number of shaving attempts, and the threshold is set to 57%.

Figure 5.6 shows the results: recall that although the plots in Figure 5.4 and the correlation coefficient were done only on a subset of the instances (not too easy and no too hard and with a reasonable amount of quick shaving), here we present the average gain or loss over all 630 instances. QS with the shaving filter adds almost no running time overhead to any of the problem classes / heuristics; in the best case it behaves very similarly to pure QS. Despite just a moderate correlation, IBS took advantage of the shaving filter and was able to improve the results over QS in all the problem classes; nonetheless, the shaving filter was not able to completely clear the overhead for the market split problem but it considerably reduced it passing from a time ratio of 1.80 to 1.15.

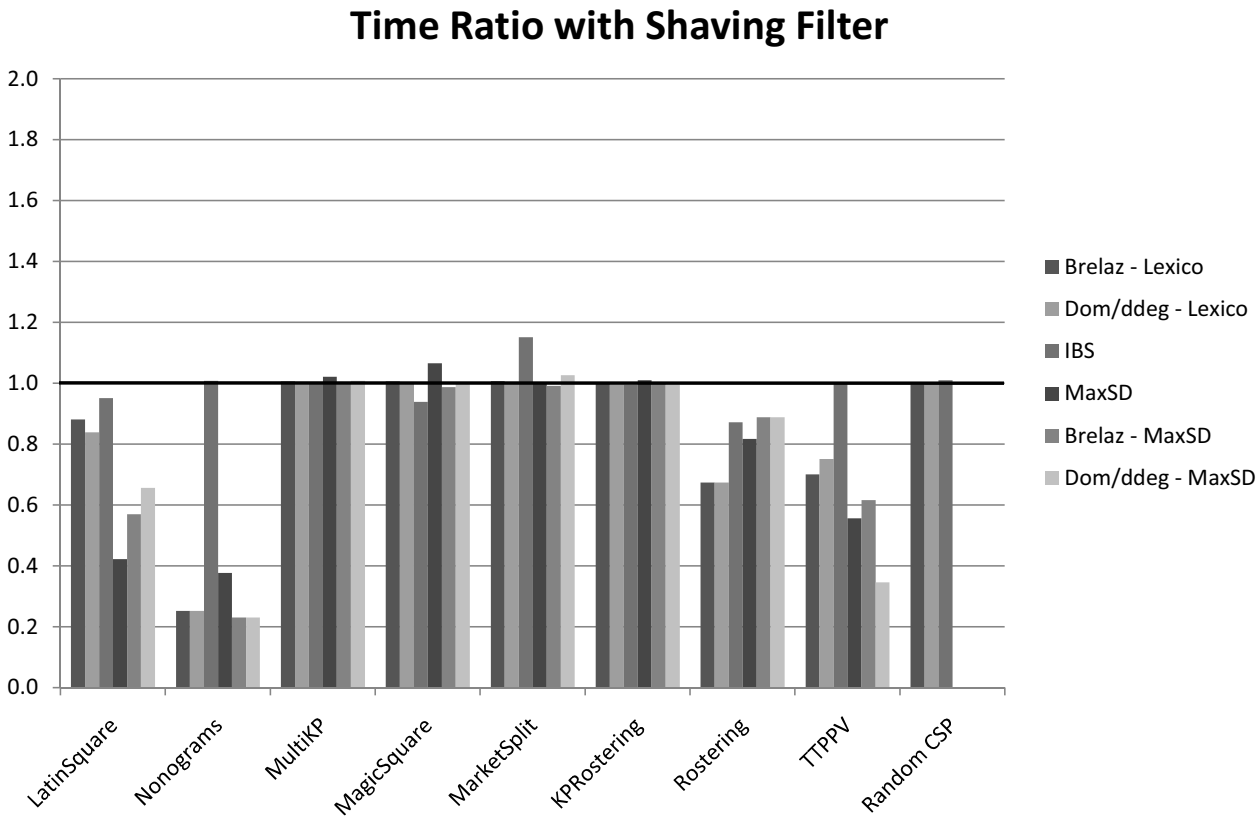


Figure 5.6 Time Ratio: average time of QS with shaving filter over average time without QS

5.4 Conclusion

This chapter studied the behavior of quick shaving combined with several search heuristics and applied to several problem classes. Out of the empirical data collected, a simple but telling feature emerged, allowing quick shaving to be tried with almost no computational overhead in cases where it should not and with significant computational advantage in cases where it should. This is a step forward in ease of use for the end-users of shaving procedures. The proposed procedure is very easy to implement on top of quick shaving.

This is by no means a definitive answer in understanding strengths and weaknesses of QS but only a first step: in the future, we would like to better investigate also the relationship between the reduction of the search space induced by QS and its performance. We would like also to test on new problems and new instances to verify whether the threshold we found appears in a new problem set. Finally, more elaborate techniques should be explored in order to have a more fine grained procedure to dynamically enable and disable QS during search.

CHAPTER 6

Conclusions

The studies conducted and presented in this thesis revolve principally around global constraints— they touch both main aspects of Constraint Programming, i.e. filtering and search.

For what concerns the first aspect, we studied in Chapter 2 the `gcc` constraint that is a recurrent sub-structure in real-life problems. We worked on the relaxed version of the `gcc` and we proposed a new filtering algorithm that outperforms in terms of time complexity the previously known algorithm both in the consistency check and even more importantly in the filtering. This has a direct impact as more efficient filtering translates to the possibility to tackle more difficult or larger problem. The contribution goes beyond efficiency concerns as it touches engineering and implementation aspects. The previously known algorithm, and more generally filtering algorithms based on min-cost max-flow, are rather hard to implement and, so far as we know, they are not present in any available solver. From an implementation standpoint, the algorithm proposed follows very closely the `gcc` filtering algorithm therefore the efforts for implementing it are sensibly reduced. Our algorithm has been implemented already in a commercial solver, Comet^{TM}by Dynadec, and used to solve real-life applications.

Studies on the `gcc` brought us to identify also a new sub-structure found in real-life scenarios. This new global constraint is a generalization of the `gcc` and it embeds the concept of hierarchical resources and skills directly into a single global constraint. A hard and a soft version have been proposed and we showed both theoretically and experimentally (only for the hard version of the constraint) the effectiveness of the filtering algorithm introduced. This contribution affects directly the efficacy in solving problems containing aspects related to heteregenous resource allocation.

For what concerns search, we introduced in Chapter 3 efficient counting algorithms for the `alldifferent`, the `symmetric_alldifferent`, the `gcc` and the `regular` constraints. We examined experimentally the quality of the approximation of the solution count for the `alldifferent` constraint. These counting algorithms form the underlying infrastructure for a family of heuristics, introduced in Chapter 4, that base their branching strategies on the number of solutions of the constraints. These heuristics are of course far from being flawless, nonetheless they showed very promising results over eight different problems. The percentage

of instances solved is not rivaled by any other tested heuristics (among which some are considered state-of-the-art); the size of search trees and running times are on average substantially better than previously known heuristics, demonstrating therefore that the overhead introduced by counting algorithms can pay off. Counting algorithms and the ideas introduced in counting-based heuristics can be already used as basic building blocks for highly efficient custom search heuristics. However, at the same time, a transparent, systematic and automated use of counting-based heuristics raises some questions and issues that must be addressed before these heuristics can be actually adopted. Particularly: for the moment, the availability of counting algorithms for a limited number of constraints narrows the applicability to only some problem domains; branching is based on individual constraints therefore it might miss opportunities that a broader view may capture; counting information on different variable sets might render difficult comparisons of solution densities; and last but not least solving optimization problems has yet to be addressed. Nonetheless, for each one of these points we gave already some insights on how they could be addressed in future studies. Overall counting-based heuristics are significant step toward a completely automated, generic and yet efficient search procedure in CP. This has a potential impact on broadening the use of CP to solve real-life problems.

Finally in Chapter 5, we studied experimentally a well-known technique, quick shaving, to achieve a reduced form of singleton consistency. This strong form of filtering, when performed systematically, might incur a significant overhead — nonetheless on some specific problems it brings clear benefits. We introduced a simple yet very effective estimator to dynamically disable quick shaving and showed experimentally very promising results. Our procedure let quick shaving speed up the solving process every time it produces some benefits, and at the same time, it disabled it for each instance for which it is counterproductive. Interestingly, the procedure does not add any significant overhead. The improved quick shaving will possibly widen the use of shaving techniques by lifting the end-user the burden of fine tuning the consistency level of solvers. Yet some improvements can be foreseen in this research: first of all we should consider also the amount of domain reductions and not only the percentage of successful shaving attempts. Secondly, more in-depth studies on the trends of the shaving ratio on a given instance could give better insights into the quick shaving procedure and on how/when to disable it. Finally, a procedure that allows to dynamically enable/disable quick shaving depending on the current subtree that is being explored could possibly give better results than the procedure proposed.

Bibliography

[1] AHUJA, R. K., MAGNANTI, T. L. et ORLIN, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall.

[2] ALON, N. et FRIEDLAND, S. (2008). The Maximum Number of Perfect Matchings in Graphs with a Given Degree Sequence. *The Electronic Journal of Combinatorics*, 15, N13.

[3] BALAFOUTIS, T. et STERGIOU, K. (2008). Experimental evaluation of modern variable selection strategies in Constraint Satisfaction Problems. *Proceedings of the Fifteenth Knowledge Representation and Automated Reasoning Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA-08*.

[4] BALAFOUTIS, T. et STERGIOU, K. (2008). On Conflict-driven variable ordering heuristics. *Proceedings of Thirteenth Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP-08*.

[5] BARTÁK, R. et ERBEN, R. (2004). A New Algorithm for Singleton Arc Consistency. *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, FLAIRS-04*. AAAI Press.

[6] BARVINOK, A. (1999). Polynomial time algorithms to approximate permanents and mixed discriminants within a simply exponential factor. *Random Structures & Algorithms*, 14, 29–61.

[7] BECK, J. C. et PERRON, L. (2000). Discrepancy-Bounded Depth First Search. *Proceedings of the Second International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR-2000*. 133–147.

[8] BECK, J. C., PROSSER, P. et WALLACE, R. J. (2004). Trying Again to Fail-First. *Recent Advances in Constraints*. Springer, vol. 3419 de *LNAI*, 41–55.

[9] BELDICEANU, N. et DEMASSEY, S. (last consulted 2010-03). Global constraint catalog. http://www.emn.fr/x-info/sdemasse/gccat/biblio.html.

[10] BESSIÈRE, C., CHMEISS, A. et SAIS, L. (2001). Neighborhood-Based Variable Ordering Heuristics for the Constraint Satisfaction Problem. *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming, CP-01*. Springer-Verlag, vol. 2239 de *LNCS*, 565–569.

[11] BESSIÈRE, C. et DEBRUYNE, R. (2005). Optimal and Suboptimal Singleton Arc Consistency Algorithms. *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, IJCAI-05*. 54–59.

[12] BESSIÈRE, C. et RÉGIN, J.-C. (1996). Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, CP-96*. 61–75.

[13] BESSIÈRE, C. et RÉGIN, J.-C. (1996). MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, CP-96*. Springer Berlin / Heidelberg, vol. 1118 de *LNCS*, 61–75.

[14] BOUSSEMART, F., HEMERY, F., LECOUTRE, C. et SAIS, L. (2004). Boosting Systematic Search by Weighting Constraints. *Proceedings of the Sixteenth Eureopean Conference on Artificial Intelligence, ECAI-04*. IOS Press, 146–150.

[15] BRAS, R. L., ZANARINI, A. et PESANT, G. (2009). Efficient Generic Search Heuristics within the EMBP framework. *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming, CP-04*. Springer, vol. 5732 de *LNCS*, 539–553.

[16] BREGMAN, L. M. (1973). Some Properties of Nonnegative Matrices and their Permanents. *Soviet Mathematics Doklady*, 14, 945–949.

[17] BRÉLAZ, D. (1979). New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22, 251–256.

[18] BRODER, A. (1986). How Hard Is It to Marry at Random? (On the approximation of the permanent). *Proceedings of the eighteenth annual ACM symposium on Theory of computing, STOC-86*. ACM, 50–58. Erratum in Proceedings of the 20th Annual ACM Symposium on Theory of Computing, p. 551, 1988.

[19] CHEN, P. Y. et POPOVICH, P. M. (2002). *Correlation. Parametric and Nonparametric Measures*. Sage Publications, Inc.

[20] CHIEN, S., RASMUSSEN, L. E. et SINCLAIR, A. (2002). Clifford algebras and approximating the permanent. *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, STOC-02*. ACM, 222–231.

[21] CORNUÉJOLS, G. et DAWANDE, M. (1999). A Class of Hard Small 0-1 Programs. *INFORMS Journal of Computing*, 11, 205–210.

[22] CORREIA, M. et BARAHONA, P. (2007). On the Integration of Singleton Consistencies and Look-Ahead Heuristics. *Proceedings of the Twelfth Annual ERCIM Interna-*

tional Workshop on Constraint Solving and Constraint Logic Programming, CSCLP-07. Springer, vol. LNCS 5129, 62–75.

[23] CORREIA, M. et BARAHONA, P. (2008). On the Efficiency of Impact Based Heuristics. *Proceedings of the Fourteenth International Conference on Principles and Practice of Constraint Programming, CP-08*. Springer, vol. LNCS 5202, 608–612.

[24] DEBRUYNE, R. et BESSIÈRE, C. (1997). Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI-97*. Morgan Kaufmann, 412–417.

[25] DECHTER, R. et PEARL, J. (1987). Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34, 1–38.

[26] DEVILLE, Y. et HENTENRYCK, P. V. (1991). An efficient arc consistency algorithm for a class of csp problems. *Proceedings of the 12th International Joint conference on Artificial intelligence, IJCAI-91*. Morgan Kaufmann Publishers Inc., 325–330.

[27] DUBOIS, D., FARGIER, H. et PRADE, H. (1993). The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. *Proceedings of the Second IEEE International Conference on Fuzzy Systems, FUZZ-IEEE-93*. vol. 2, 1131–1136.

[28] ELBASSIONI, K., KATRIEL, I., KUTZ, M. et MAHAJAN, M. (2005). Simultaneous Matchings. *Proceedings of the Sixteenth Annual International Symposium on Algorithms and Computation, ISAAC-05*. Springer, vol. LNCS 3827, 106–115.

[29] FARGIER, H., LANG, J. et SCHIEX, T. (1993). Selecting preferred solutions in fuzzy constraint satisfaction problems. *Proceedings of the First European Congress on Fuzzy and Intelligent Technologies, EUFIT 93*. vol. 3, 1128–1134.

[30] FREVILLE, A. et PLATEAU, G. (1990). A Branch and Bound Method for the Multiconstraint Zero One Knapsack Problem. *Investigation Operativa*, 1, 251–270.

[31] FRIEDLAND, S. (2008). An Upper Bound for the Number of Perfect Matchings in Graphs. http://arxiv.org/abs/0803.0864.

[32] FÜRER, M. et KASIVISWANATHAN, S. P. (2004). An Almost Linear Time Approximation Algorithm for the Permanent of a Random (0-1) Matrix. *LNCS 3258*. Springer-Verlag, 54–61.

[33] GENT, I., PROSSER, P. et WALSH, T. (1996). The Constrainedness of Search. *Proceedings of Thirteenth National Conference on Artificial Intelligence, AAAI-96*. AAAI Press / The MIT Press, 246–252.

[34] GENT, I. P., MACINTYRE, E., PROSSER, P., SMITH, B. M. et WALSH, T. (1996). An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. *Proceedings of the Second International Conference on Principles and*

Practice of Constraint Programming, CP-96. Springer Berlin / Heidelberg, vol. 1118 de *LNCS*, 179–193.

[35] GENT, I. P., MACINTYRE, E., PROSSER, P., SMITH, B. M. et WALSH, T. (1996). An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, CP-96*. Springer Berlin / Heidelberg, vol. 1118 de *LNCS*, 179–193.

[36] GENT, I. P., MACINTYRE, E., PROSSER, P., SMITH, B. M. et WALSH, T. (2001). Random Constraint Satisfaction: Flaws and Structure. *Constraints*, 6, 345–372.

[37] GENT, I. P., WALSH, T., HNICH, B. et MIGUEL, I. (last consulted 2009-08). A Problem Library for Constraints. http://www.csplib.org.

[38] GODSIL, C. et GUTMAN, I. (1981). On the matching polynomial of a graph. *Algebraic Methods in Graph Theory*, 241–249.

[39] GOGATE, V. et DECHTER, R. (2006). A New Algorithm for Sampling CSP Solutions Uniformly at Random. *Proceedings of Twelfth International Conference on Principles and Practice of Constraint Programming, CP-06*. 711–715.

[40] GOGATE, V. et DECHTER, R. (2007). Approximate Counting by Sampling the Backtrack-free Search Space. *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, AAAI-07*. AAAI Press, 198–203.

[41] GOGATE, V. et DECHTER, R. (2008). Studies in Solution Sampling. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI-07*. AAAI Press, 271–276.

[42] GOMES, C., HOFFMANN, J., SABHARWAL, A. et SELMAN, B. (2007). From Sampling to Model Counting. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence, IJCAI-07*. 2293–2299.

[43] GOMES, C., SABHARWAL, A. et SELMAN, B. (2006). Model counting: A new strategy for obtaining good bounds. *Proceedings The Twenty-First National Conference on Artificial Intelligence, AAAI-06*. AAAI Press, 54–61.

[44] GOMES, C., SELMAN, B. et CRATO, N. (1997). Heavy-tailed Distributions in Combinatorial Search. *Proceedings of Third International Conference on Principles and Practice of Constraint Programming, CP-97*. Springer, vol. LNCS 1330, 121–135.

[45] GOMES, C., SELMAN, B. et KAUTZ, H. (1998). Boosting Combinatorial Search Through Randomization. *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence, AAAI-98/IAAI-98*. AAAI Press, 431–437.

[46] GOMES, C. et SHMOYS, D. (2002). Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. *Proceedings of Computational Symposium on Graph Coloring and Generalizations, COLOR-02*. 22–39.

[47] GOMES, C., VAN HOEVE, W., SABHARWAL, A. et SELMAN, B. (2007). Counting CSP solutions using generalized XOR constraints. *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, AAAI-07*. AAAI Press, 204–209.

[48] GRIMES, D. et WALLACE, R. J. (2006). Learning to Identify Global Bottlenecks in Constraint Satisfaction Search. *Learning for Search: Papers from the AAAI-06 Workshop*. vol. Tech. Rep. WS-06-11, 24–31.

[49] GRIMES, D. et WALLACE, R. J. (2007). Sampling Strategies and Variable Selection in Weighted Degree Heuristics. *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming, CP-07*. Springer, vol. 4741 de *LNCS*, 831–838.

[50] H. J. RYSER (1963). *Combinatorial Mathematics*, vol. 14 de *Carus Mathematical Monographs*. Mathematical Association of America, John Wiley and Sons.

[51] HARALICK, R. M. et ELLIOTT, G. L. (1980). Increasing Tree Seach Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14, 263–313.

[52] HARVEY, W. D. et GINSBERG, M. L. (1995). Limited Discrepancy Search. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI-95*. Morgan Kaufmann, 607–615.

[53] HOPCROFT, J. E. et KARP, R. M. (1973). An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2, 225–231.

[54] HSU, E. I., KITCHING, M., BACCHUS, F. et MCILRAITH, S. A. (2007). Using Expectation Maximization to Find Likely Assignments for Solving CSP's. *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*. AAAI Press, 224–230.

[55] HUBER, M. (2006). Exact Sampling from Perfect Matchings of Dense Regular Bipartite Graphs. *Algorithmica*, 44, 183–193.

[56] HULUBEI, T. et O'SULLIVAN, B. (2006). The Impact of Search Heuristics on Heavy-Tailed Behaviour. *Constraints*, 11, 159–178.

[57] JERRUM, M., SINCLAIR, A. et VIGODA, E. (2001). A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries. *Proceedings of the thirty-third annual ACM symposium on Theory of computing, STOC-01*. ACM, 712–721.

[58] JERRUM, M., SINCLAIR, A. et VIGODA, E. (2004). A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries. *Journal ACM*, 51, 671–697.

[59] JERRUM, M. R., VALIANT, L. G. et VAZIRANI, V. V. (1986). Random generation of combinatorial structures from a uniform. *Theoretical Computer Science*, 43, 169–188.

[60] JURKAT, W. et RYSER, H. J. (1966). Matrix Factorizations of Determinants and Permanents. *Journal of Algebra*, 3, 1–27.

[61] KASK, K., DECHTER, R. et GOGATE, W. (2004). Counting-Based Look-Ahead Schemes for Constraint Satisfaction. Springer-Verlag, éditeur, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP-04*. vol. LNCS 3258, 317–331.

[62] KASTELEYN, P. (1961). The Statistics of Dimers on a Lattice. *Physica*, 27, 1209 – 1225.

[63] KORF, R. E. (1985). Depth-First Iterative-Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence*, 27, 97–109.

[64] LARROSA, J. (2002). Node and Arc Consistency in Weighted CSP. *Proceedings of the Eighteenth National Conference on Artificial Intelligence, AAAI-02*. AAAI Press, 48–53.

[65] LARROSA, J. et SCHIEX, T. (2003). In the Quest of the best form of local consistency for Weighted CSP. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI-03*. Morgan Kaufmann, 239–244.

[66] LHOMME, O. (1993). Consistency Techniques for Numeric CSPs. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, IJCAI-93*. Morgan Kaufmann, 232–238.

[67] LHOMME, O. (2005). Quick Shaving. *Proceedings of the Twentieth AAAI Conference on Artificial Intelligence, AAAI-05*. AAAI Press / The MIT Press, 411–415.

[68] LIANG, H. et BAI, F. (2004). An Upper Bound for the Permanent of (0,1)-Matrices. *Linear Algebra and its Applications*, 377, 291–295.

[69] LINIAL, N., SAMORODNITSKY, A. et WIGDERSON, A. (2000). A Deterministic Strongly Polynomial Algorithm for Matrix Scaling and Approximate Permanents. *Combinatorica*, 20, 545–568.

[70] LUBY, M., SINCLAIR, A. et ZUCKERMAN, D. (1993). Optimal Speedup of Las Vegas Algorithms. *Information Processing Letters*, 47, 173–180.

[71] M. JERRUM (2003). *Counting, Sampling And Integrating: Algorithms And Complexity*. Birkhäuser Basel.

[72] MARTIN, P. et SHMOYS, D. B. (1996). A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem. *Proceedings of the Fifth International on Integer Programming and Combinatorial Optimization, IPCO-96*. Springer, 389–403.

[73] MEISELS, A., SHIMONY, S. E. et SOLOTOREVSKY, G. (2000). Bayes Networks for Estimating the Number of Solutions of Constraint Networks. *Annals of Mathematics and Artificial Intelligence*, 28, 169–186.

[74] MELO, R., URRUTIA, S. et RIBEIRO, C. (2009). The traveling tournament problem with predefined venues. *Journal of Scheduling*, 12, 607–622.

[75] MÉTIVIER, J.-P., BOIZUMAULT, P. et LOUDNI, S. (2009). Softening gcc and regular with preferences. *Proceedings of the 2009 ACM Symposium on Applied Computing, SAC-09*. ACM, 1392–1396.

[76] MINC, H. (1963). Upper Bounds for Permanents of (0, 1)-matrices. *Bulletin of the American Mathematical Society*, 69, 789–791.

[77] PESANT, G. (2004). A Regular Language Membership Constraint for Finite Sequences of Variables. *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP-04*. Springer, vol. 3258 de *LNCS*, 482–495.

[78] PESANT, G. (2005). Counting solutions of csps: A structural approach. *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, IJCAI-05*. 260–265.

[79] PESANT, G. et QUIMPER, C.-G. (2008). Counting solutions of knapsack constraints. *Proceedings of the Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR-08*. 203–217.

[80] PETERSEN, J. (1891). Die Theorie der Regulâren Graphs. *Acta Mathematica*, 15, 193–220.

[81] PETIT, T., RÉGIN, J.-C. et BESSIÈRE, C. (2001). Specific Filtering Algorithms for Over Constrained Problems. *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming, CP-01*. Springer, vol. LNCS 2239, 451–463.

[82] PROSSER, P., STERGIOU, K. et WALSH, T. (2002). Singleton Consistencies. *Proceedings of the Eighth International Conference Principles and Practice of Constraint Programming, CP-02*. Springer-Verlag, vol. LNCS 2470, 353–368.

[83] QUIMPER, C., LOPEZ-ORTIZ, A., VAN BEEK, P. et GOLYNSKI, A. (2004). Improved algorithms for the global cardinality constraint. *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP-04*. Springer, vol. LNCS 3258, 542–556.

[84] R. J. BAYARDO, J. et PEHOUSHEK, J. D. (2000). Counting Models Using Connected Components. *Proceedings of the Seventeenth National Conference on Artificial Intelligence, AAAI-00*. AAAI Press / The MIT Press, 157–162.

[85] RASMUSSEN, L. E. (1994). Approximating the Permanent: A Simple Approach. *Random Structures Algorithms*, 5, 349–362.

[86] REFALO, P. (2004). Impact-Based Search Strategies for Constraint Programming. *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP-04*. Springer, vol. LNCS 3258, 557–571.

[87] RÉGIN, J. (1999). The Symmetric Alldiff Constraint. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI-99*. Morgan Kaufmann Publishers Inc., 420–425.

[88] RÉGIN, J.-C. (1994). A Filtering Algorithm for Constraints of Difference in CSPs. *Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI-94*. American Association for Artificial Intelligence, vol. 1, 362–367.

[89] RÉGIN, J.-C. (1996). Generalized Arc Consistency for Global Cardinality Constraint. *Proceedings of the Thirteenth National/Eighth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI-98/IAAI-98*. vol. 1, 209–215.

[90] RÉGIN, J.-C. (1999). Arc Consistency for Global Cardinality Constraints with Costs. *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming, CP-99*. Springer-Verlag, vol. LNCS 1713, 390–404.

[91] ROADEF, C. (last consulted 2010-03). http://www.g-scop.inpg.fr/ChallengeROADEF2007/.

[92] ROSSI, F., BEEK, P. V. et WALSH, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA.

[93] SABIN, D. et FREUDER, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP-94*. Springer-Verlag, 10–20.

[94] SANG, T., BACCHUS, F., BEAME, P., KAUTZ, H. A. et PITASSI, T. (2004). Combining Component Caching and Clause Learning for EffectiveModel Counting. *Proceed-*

ings of Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT-04.

[95] SCHIEX, T. (1992). Possibilistic Constraint Satisfaction Problems or "How to handle soft constraints?". *Proceedings of the Eighth Annual Conference on Uncertainty in Artificial Intelligence, UAI-92*. Morgan Kaufmann, 268–275.

[96] SHI, W. (1979). A Branch and Bound Method for the Multiconstraint Zero One Knapsack Problem. *Journal of the Operational Research Society*, 30, 369–378.

[97] SMITH, B. M. et GRANT, S. A. (1998). Trying Harder to Fail First. *Thirteenth European Conference on Artificial Intelligence, ECAI-98*. John Wiley & Sons, 249–253.

[98] SOULES, G. W. (2003). New Permanental Upper Bounds for Nonnegative Matrices. *Linear and Multilinear Algebra*, 51, 319–337.

[99] SOULES, G. W. (2005). Permanental Bounds for Nonnegative Matrices via Decomposition. *Linear Algebra and its Applications*, 394, 73–89.

[100] STAMATATOS, E. et STERGIOU, K. (2009). Learning How to Propagate Using Random Probing. *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR-09*. Springer-Verlag, 263–278.

[101] SZYMANEK, R. et LECOUTRE, C. (2008). Constraint-Level Advice for Shaving. *Proceedings of the Twenty-fourth International Conference on Logic Programming, ICLP-08*. Springer-Verlag, 636–650.

[102] SZYMANEK, R. et O'SULLIVAN, B. (2006). Guiding Search using Constraint-Level Advice. *Proceeding of Seventeenth European Conference on Artificial Intelligence, ECAI-06*. IOS Press, vol. 141 de *Frontiers in Artificial Intelligence and Applications*, 158–162.

[103] TARJAN, R. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1, 146–160.

[104] TRICK, M. A. (2003). A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118, 73–84.

[105] VALIANT, L. (1979). The Complexity of Computing the Permanent. *Theoretical Computer Science*, 8, 189–201.

[106] VAN HOEVE, W. J. (2001). The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015.

[107] VAN HOEVE, W. J., PESANT, G. et ROUSSEAU, L.-M. (2006). On global warming: Flow-based soft global constraints. *Journal of Heuristics*, 12, 347–373.

[108] WALSH, T. (1997). Depth-Bounded Discrepancy Search. *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence, IJCAI-97*. Morgan Kaufmann Publishers Inc., 1388–1393.

[109] WALSH, T. (1999). Search in a Small World. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI-99*. 1172–1177.

[110] WASSERMANN, A. (last consulted 2007-11). The feasibility version of the market split problem. http://did.mat.uni-bayreuth.de/ alfred/marketsplit.html.

[111] ZANARINI, A., MILANO, M. et PESANT, G. (2006). Improved algorithm for the soft global cardinality constraint. *Proceedings of the Third International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR-06*. Springer, vol. LNCS 3990, 288–299.

[112] ZANARINI, A. et PESANT, G. (2007). Generalizations of the global cardinality constraint for hierarchical resources. *Proceedings of the Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR-07*. Springer, vol. LNCS 4510, 361–375.

[113] ZANARINI, A. et PESANT, G. (2007). Solution Counting Algorithms for Constraint-Centered Search Heuristics. *CP*. 743–757.

[114] ZANARINI, A. et PESANT, G. (2009). Solution Counting Algorithms for Constraint-Centered Search Heuristics. *Constraints*. vol. 14, 392–413.

[115] ZANARINI, A. et PESANT, G. (2009). When Can I Get a Quick Shave? *Proceedings of the Eighth International Workshop on Constraint Modelling and Reformulation, ModRef-09*. 186–200.

[116] ZANARINI, A. et PESANT, G. (2010). More Robust Counting-Based Search Heuristics with Alldifferent Constraints. *to appear in CPAIOR-2010*.

Full Experimental Results

Next, we report the full list of experimental results for the eight problem classes reported in Chapter 4. Columns show respectively:

- *a.T(S)* - arithmetic average of time for solved instances (in seconds)
- *a.T* - arithmetic average of time for solved and timeout instances (in seconds)
- *a.Bcks* - arithmetic average of number of backtracks for solved and timeout instances
- *g.T* - geometric average of time for solved and timeout instances (in seconds)
- *g.Bcks* - geometric average of number of backtracks for solved and timeout instances
- *% sol* - percentage of solved instances

N/A is reported when the result does not apply, or when numerical overflow occured.

Randomized heuristics have been run 10 times on each instance and the average has been considered. All the tests have been performed on a AMD Opteron 2.2GHz with 1GB and Ilog Solver 6.6. Please refer to Section 4.3 for further explanation of the experimental setup and for the heuristic tested.

Table .1 Average results for 40 hard Latin Square instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	301.9	658.2	1300056	501.5	1001608	56.75%
Brélaz; lexico	254.0	750.6	1458276	432.6	846512	47.50%
dom/ddeg; lexico	333.8	723.6	1387456	404.9	780784	55.00%
IBS	125.1	716.3	900253	256.9	345228	45.00%
IBS+	335.0	532.9	1595756	273.9	826433	71.25%
Ilog IBS+	194.0	344.9	914849	91.1	247775	85.00%
RSC+LA	305.8	350.5	856	158.8	291	95.00%
RSC2+LA	125.8	179.5	4880	29.8	0	95.00%
Brélaz; maxSD	136.2	322.4	569170	38.1	65609	82.50%
dom/ddeg; maxSD	134.6	294.5	526034	46.7	85479	85.00%
IBS; maxSD	209.2	432.1	493791	154.8	188100	77.50%
IBS+; maxSD	207.1	430.5	493844	154.0	188111	77.50%
maxSD	77.1	105.2	104672	8.5	7512	97.50%
maxAggr(maxRelSD)	84.3	195.9	212064	17.5	19203	90.00%
maxAggr(maxRelRatio)	554.2	1022.4	1109879	858.5	933279	27.50%
maxAggr(min)	124.8	286.1	316127	28.0	30791	85.00%
maxAggr(aAvg)	138.6	191.6	207699	19.1	19372	95.00%
maxAggr(wSCAvg)	91.4	202.2	206193	17.0	17492	90.00%
maxAggr(wAntiSCAvg)	127.9	261.9	246169	45.2	43767	87.50%
maxAggr(wTAvg)	149.2	254.3	259359	45.1	47407	90.00%
maxAggr(wAntiTAvg)	87.8	199.0	177918	16.7	15239	90.00%
maxAggr(wDAvg)	141.8	194.7	195966	29.8	31511	95.00%
maxAggr(SCRemaining)	65.9	1171.7	1226708	1116.0	1169686	2.50%
minAggr(SCRemoved)	329.0	655.7	584285	265.3	240215	62.50%
maxAggr(solProb)	162.7	681.3	633569	276.8	262331	50.00%
maxAggr(avgSDDisc)	308.3	754.1	628906	428.9	371153	50.00%
maxAggr(maxSDDisc)	305.5	752.8	634344	427.8	373168	50.00%
minSCMaxSD	257.8	917.3	1115497	528.1	642262	30.00%
minTMaxSD	269.6	920.9	1075742	535.4	624663	30.00%
maxSCMaxSD	242.1	1032.4	1458590	773.5	1097948	17.50%
maxTMaxSD	375.2	932.0	1048746	635.1	717973	32.50%
minDom; MaxSD	63.4	120.3	139965	9.8	10045	95.00%
maxAvgVar; maxSD	77.8	218.0	195997	32.3	31005	87.50%
maxRegretVar; maxSD	102.8	130.2	159349	15.0	17815	97.50%
maxMinVar; maxSD	118.9	362.2	326770	67.1	56126	77.50%

Table .2 Average results for 180 Nonogram instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	37.4	204.2	119756	0.0	0	84.83%
Brélaz; lexico	55.2	296.9	193618	0.0	0	78.89%
dom/ddeg; lexico	54.5	296.4	194889	0.0	0	78.89%
IBS	0.8	7.4	4712	0.0	0	99.44%
IBS+	5.1	6.9	12640	0.4	0	99.56%
Ilog IBS+	0.7	7.4	12668	0.0	0	99.44%
RSC+LA	2.9	2.9	10	0.5	0	100.00%
RSC2+LA	2.2	2.2	6	0.5	0	100.00%
Brélaz; maxSD	44.4	294.8	184267	0.0	0	78.33%
dom/ddeg; maxSD	44.9	295.2	186598	0.0	0	78.33%
IBS; maxSD	0.7	7.4	5785	0.0	0	99.44%
IBS+; maxSD	0.7	7.4	5707	0.0	0	99.44%
maxSD	8.7	48.5	18096	0.0	0	96.67%
maxAggr(maxRelSD)	8.9	48.6	18216	0.0	0	96.67%
maxAggr(maxRelRatio)	9.1	48.8	17774	0.0	0	96.67%
maxAggr(min)	19.8	150.9	73493	0.0	0	88.89%
maxAggr(aAvg)	22.5	120.6	57061	0.0	0	91.67%
maxAggr(wSCAvg)	16.3	82.1	29591	0.0	0	94.44%
maxAggr(wAntiSCAvg)	31.7	122.5	53947	0.0	0	92.22%
maxAggr(wTAvg)	18.5	97.2	40143	0.0	0	93.33%
maxAggr(wAntiTAvg)	3.9	77.0	27407	0.0	0	93.89%
maxAggr(wDAvg)	10.7	57.0	19782	0.0	0	96.11%
maxAggr(SCRemaining)	47.6	502.1	301203	0.0	0	60.56%
minAggr(SCRemoved)	4.6	37.8	13009	0.0	0	97.22%
maxAggr(solProb)	23.5	252.3	110447	0.0	0	80.56%
maxAggr(avgSDDisc)	11.7	18.3	5312	0.0	0	99.44%
maxAggr(maxSDDisc)	11.5	18.1	5251	0.0	0	99.44%
minSCMaxSD	3.2	29.8	19797	0.0	0	97.78%
minTMaxSD	3.3	29.9	19965	0.0	0	97.78%
maxSCMaxSD	13.3	191.3	133364	0.0	0	85.00%
maxTMaxSD	15.6	173.5	107193	0.0	0	86.67%
minDom; MaxSD	9.1	48.8	17900	0.0	0	96.67%
maxAvgVar; maxSD	3.9	17.2	5052	0.0	0	98.89%
maxRegretVar; maxSD	21.0	119.2	59075	0.0	0	91.67%
maxMinVar; maxSD	6.5	26.4	7399	0.0	0	98.33%

Table .3 Average results for 25 Multi Knapsack instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	169.3	620.9	11585	90.6	1518	51.60%
Brélaz; lexico	102.4	497.9	11212	0.0	0	64.00%
dom/ddeg; lexico	104.9	499.5	11130	0.0	0	64.00%
IBS	86.5	354.0	4080	41.6	679	76.00%
IBS+	258.9	435.0	8607	72.8	1654	75.20%
Ilog IBS+	140.7	395.6	4121	63.2	1256	76.00%
RSC+LA	204.6	603.1	450	0.0	0	60.00%
RSC2+LA	230.1	618.3	527	144.2	0	60.00%
Brélaz; maxSD	111.9	504.1	9948	0.0	0	64.00%
dom/ddeg; maxSD	107.1	501.0	9877	50.4	0	64.00%
IBS; maxSD	163.7	412.8	3179	0.0	768	76.00%
IBS+; maxSD	164.4	413.2	3242	0.0	770	76.00%
maxSD	25.7	72.7	1228	0.0	0	96.00%
maxAggr(maxRelSD)	25.4	72.4	1250	0.0	0	96.00%
maxAggr(maxRelRatio)	26.2	73.2	1245	0.0	0	96.00%
maxAggr(min)	14.2	583.6	8572	0.0	0	52.00%
maxAggr(aAvg)	6.7	6.7	29	0.0	0	100.00%
maxAggr(wSCAvg)	55.6	422.2	2198	0.0	479	68.00%
maxAggr(wAntiSCAvg)	9.4	9.4	509	0.0	0	100.00%
maxAggr(wTAvg)	57.0	423.0	2104	0.0	467	68.00%
maxAggr(wAntiTAvg)	9.9	9.9	509	0.0	0	100.00%
maxAggr(wDAvg)	6.7	6.7	29	0.0	0	100.00%
maxAggr(SCRemaining)	54.7	421.5	2125	0.0	473	68.00%
minAggr(SCRemoved)	55.7	422.1	2112	0.0	468	68.00%
maxAggr(solProb)	55.1	100.9	2687	3.3	0	96.00%
maxAggr(avgSDDisc)	138.4	308.5	978	0.0	203	84.00%
maxAggr(maxSDDisc)	116.7	333.7	1265	0.0	231	80.00%
minSCMaxSD	112.3	155.9	1769	0.0	0	96.00%
minTMaxSD	108.5	152.2	1767	9.6	0	96.00%
maxSCMaxSD	129.3	514.9	9916	0.0	0	64.00%
maxTMaxSD	129.5	515.2	9994	0.0	0	64.00%
minDom; MaxSD	25.7	72.7	1246	0.0	0	96.00%
maxAvgVar; maxSD	68.9	295.8	1482	0.0	0	80.00%
maxRegretVar; maxSD	71.8	71.8	2880	0.0	0	100.00%
maxMinVar; maxSD	99.0	407.6	4565	0.0	0	72.00%

Table .4 Average results for 10 Market Split instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	268.7	268.7	115538	257.1	110044	100.00%
Brélaz; lexico	121.6	121.6	109742	101.9	91070	100.00%
dom/ddeg; lexico	122.4	122.4	109742	102.9	91070	100.00%
IBS	202.3	202.3	90936	162.9	76601	100.00%
IBS+	584.5	688.0	358098	670.3	345023	83.00%
Ilog IBS+	620.9	736.8	420780	599.0	327518	80.00%
RSC+LA	716.3	909.8	22178	843.5	20208	60.00%
RSC2+LA	723.3	914.0	23158	849.5	21129	60.00%
Brélaz; maxSD	245.9	245.9	93213	141.2	53613	100.00%
dom/ddeg; maxSD	231.8	231.8	93213	136.5	53613	100.00%
IBS; maxSD	328.6	328.6	89893	184.2	52235	100.00%
IBS+; maxSD	338.6	338.6	89893	187.6	52235	100.00%
maxSD	256.9	256.9	59878	182.0	40944	100.00%
maxAggr(maxRelSD)	255.3	255.3	59878	182.0	40944	100.00%
maxAggr(maxRelRatio)	253.7	253.7	59878	181.5	40944	100.00%
maxAggr(min)	424.2	424.2	118705	305.3	86016	100.00%
maxAggr(aAvg)	274.8	274.8	69251	176.5	43986	100.00%
maxAggr(wSCAvg)	307.0	307.0	74505	191.8	45355	100.00%
maxAggr(wAntiSCAvg)	276.1	276.1	66975	171.7	40957	100.00%
maxAggr(wTAvg)	304.4	304.4	74505	190.2	45355	100.00%
maxAggr(wAntiTAvg)	270.9	270.9	66975	170.5	40957	100.00%
maxAggr(wDAvg)	280.2	280.2	69251	182.2	43986	100.00%
maxAggr(SCRemaining)	301.8	301.8	74505	190.1	45355	100.00%
minAggr(SCRemoved)	314.6	314.6	74511	192.5	45360	100.00%
maxAggr(solProb)	285.6	285.6	72570	185.9	45892	100.00%
maxAggr(avgSDDisc)	175.8	175.8	37852	109.5	22928	100.00%
maxAggr(maxSDDisc)	285.4	285.4	59080	207.6	41772	100.00%
minSCMaxSD	210.6	210.6	50944	167.4	39812	100.00%
minTMaxSD	209.6	209.6	50944	164.8	39812	100.00%
maxSCMaxSD	364.6	364.6	89743	280.0	68544	100.00%
maxTMaxSD	376.3	376.3	89743	284.5	68544	100.00%
minDom; MaxSD	247.7	247.7	59878	178.0	40944	100.00%
maxAvgVar; maxSD	236.4	236.4	50565	139.9	29877	100.00%
maxRegretVar; maxSD	343.5	343.5	83327	258.2	62059	100.00%
maxMinVar; maxSD	386.9	386.9	83541	324.4	71108	100.00%

Table .5 Average results for 40 Magic Square instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	59.4	140.4	29665	72.0	14436	93.00%
Brélaz; lexico	58.9	658.7	72877	152.0	14438	47.50%
dom/ddeg; lexico	36.5	736.0	83113	197.8	21024	40.00%
IBS	73.1	638.5	24447	279.3	12679	50.00%
IBS+	644.0	716.3	43107	499.0	28449	72.00%
Ilog IBS+	551.0	632.4	20471	357.2	11658	85.25%
RSC+LA	626.8	985.7	39	926.3	0	37.50%
RSC2+LA	219.4	734.9	897	420.5	0	47.50%
Brélaz; maxSD	34.3	34.3	10592	6.3	394	100.00%
dom/ddeg; maxSD	38.3	38.3	11695	6.2	473	100.00%
IBS; maxSD	296.7	296.7	5282	178.4	4643	100.00%
IBS+; maxSD	302.2	302.2	5282	182.5	4643	100.00%
maxSD	14.1	14.1	1685	8.1	203	100.00%
maxAggr(maxRelSD)	87.7	338.9	97015	46.7	5097	77.50%
maxAggr(maxRelRatio)	669.3	1094.6	386437	918.9	271307	20.00%
maxAggr(min)	42.8	129.6	31350	30.6	2285	92.50%
maxAggr(aAvg)	22.3	51.8	10731	8.7	151	97.50%
maxAggr(wSCAvg)	24.5	672.5	29989	131.1	7956	45.00%
maxAggr(wAntiSCAvg)	11.9	41.6	11998	8.4	273	97.50%
maxAggr(wTAvg)	26.3	143.7	27924	23.7	1383	90.00%
maxAggr(wAntiTAvg)	10.5	10.5	1290	6.0	98	100.00%
maxAggr(wDAvg)	33.0	676.8	31259	131.3	8606	45.00%
maxAggr(SCRemaining)	26.5	672.8	30129	129.3	7597	45.00%
minAggr(SCRemoved)	53.5	629.0	30012	140.9	10242	50.00%
maxAggr(solProb)	170.0	558.2	59225	106.9	11160	62.50%
maxAggr(avgSDDisc)	66.5	123.2	30350	21.5	1013	95.00%
maxAggr(maxSDDisc)	22.6	199.4	101400	22.8	866	85.00%
minSCMaxSD	59.0	259.0	62575	39.9	4992	82.50%
minTMaxSD	58.4	258.4	63368	35.7	5004	82.50%
maxSCMaxSD	22.7	672.1	33073	125.7	9035	45.00%
maxTMaxSD	88.2	450.1	77610	64.0	6804	67.50%
minDom; MaxSD	11.2	11.2	1090	6.4	0	100.00%
maxAvgVar; maxSD	24.6	24.6	3564	8.7	114	100.00%
maxRegretVar; maxSD	105.5	681.2	174227	199.6	24446	47.50%
maxMinVar; maxSD	15.8	164.8	29337	14.2	445	87.50%

Table .6 Average results for 10 Cost Rostering instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	128.7	756.4	722226	88.5	53973	39.00%
Brélaz; lexico	158.7	679.4	995657	46.1	0	50.00%
dom/ddeg; lexico	162.8	681.4	1004048	42.6	0	50.00%
IBS	51.3	510.8	264037	62.9	31071	60.00%
IBS+	155.3	745.1	898336	206.5	258157	41.00%
Ilog IBS+	39.1	503.4	616198	51.2	50695	60.00%
RSC+LA	434.4	1046.9	8386	796.8	6134	20.00%
RSC2+LA	4.5	482.7	33408	37.4	0	60.00%
Brélaz; maxSD	110.0	655.0	743228	66.4	33530	50.00%
dom/ddeg; maxSD	114.9	657.5	748973	65.1	33769	50.00%
IBS; maxSD	13.5	488.1	233674	43.6	20485	60.00%
IBS+; maxSD	13.5	488.1	232443	43.8	20491	60.00%
maxSD	0.3	0.3	5	0.3	0	100.00%
maxAggr(maxRelSD)	0.3	0.3	0	0.3	0	100.00%
maxAggr(maxRelRatio)	0.3	0.3	1	0.3	0	100.00%
maxAggr(min)	238.5	719.3	498656	77.9	0	50.00%
maxAggr(aAvg)	0.3	120.3	60019	0.7	0	90.00%
maxAggr(wSCAvg)	0.3	0.3	3	0.2	0	100.00%
maxAggr(wAntiSCAvg)	0.3	120.3	46459	0.7	0	90.00%
maxAggr(wTAvg)	0.3	240.3	90582	1.7	0	80.00%
maxAggr(wAntiTAvg)	0.3	0.3	12	0.3	0	100.00%
maxAggr(wDAvg)	0.3	0.3	2	0.3	0	100.00%
maxAggr(SCRemaining)	0.3	720.1	531565	41.5	8664	40.00%
minAggr(SCRemoved)	170.7	479.5	198878	15.2	0	70.00%
maxAggr(solProb)	27.9	848.4	502298	264.8	120615	30.00%
maxAggr(avgSDDisc)	11.6	130.4	59827	3.2	0	90.00%
maxAggr(maxSDDisc)	11.6	130.4	61224	3.8	0	90.00%
minSCMaxSD	224.0	712.0	678704	103.9	46473	50.00%
minTMaxSD	231.7	715.9	684613	108.2	46755	50.00%
maxSCMaxSD	0.6	720.2	601551	49.4	20255	40.00%
maxTMaxSD	12.6	843.8	735318	207.4	197510	30.00%
minDom; MaxSD	0.2	120.2	86824	0.3	0	90.00%
maxAvgVar; maxSD	0.3	120.3	65074	0.8	0	90.00%
maxRegretVar; maxSD	0.3	0.3	0	0.3	0	100.00%
maxMinVar; maxSD	197.3	798.9	556832	103.8	0	40.00%

Table .7 Average results for 60 Rostering instances

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	12.0	231.0	2179673	25.9	0	81.50%
Brélaz; lexico	0.3	120.3	1302799	0.1	0	90.00%
dom/ddeg; lexico	0.3	120.3	1318698	0.0	0	90.00%
IBS	0.9	0.9	856	0.9	853	100.00%
IBS+	1.3	1.3	1838	1.3	1836	100.00%
Ilog IBS+	1.3	1.3	1854	1.2	1851	100.00%
RSC+LA	10.0	10.0	0	9.4	0	100.00%
RSC2+LA	159.0	853.0	220248	143.8	0	33.33%
Brélaz; maxSD	150.1	937.5	6647932	292.9	0	25.00%
dom/ddeg; maxSD	148.1	954.6	6770603	337.4	N/A	23.33%
IBS; maxSD	0.9	0.9	868	0.9	864	100.00%
IBS+; maxSD	0.9	0.9	868	0.9	864	100.00%
maxSD	5.6	164.8	416609	0.7	0	86.67%
maxAggr(maxRelSD)	0.1	0.1	0	0.1	0	100.00%
maxAggr(maxRelRatio)	0.1	20.1	61331	0.1	0	98.33%
maxAggr(min)	98.4	796.1	4048060	81.5	0	36.67%
maxAggr(aAvg)	50.6	452.9	1482255	9.4	0	65.00%
maxAggr(wSCAvg)	0.4	60.3	139299	0.2	0	95.00%
maxAggr(wAntiSCAvg)	77.0	863.1	3541720	138.7	0	30.00%
maxAggr(wTAvg)	162.1	871.4	3983134	174.8	0	31.67%
maxAggr(wAntiTAvg)	5.6	45.4	296052	0.1	0	96.67%
maxAggr(wDAvg)	0.1	20.1	39088	0.1	0	98.33%
maxAggr(SCRemaining)	2.5	22.5	64686	0.1	0	98.33%
minAggr(SCRemoved)	65.0	802.7	3107852	129.7	0	35.00%
maxAggr(solProb)	13.8	13.8	7558	0.0	0	100.00%
maxAggr(avgSDDisc)	0.1	0.1	0	0.1	0	100.00%
maxAggr(maxSDDisc)	0.1	0.1	0	0.1	0	100.00%
minSCMaxSD	0.1	0.1	191	0.1	0	100.00%
minTMaxSD	0.1	0.1	191	0.0	0	100.00%
maxSCMaxSD	3.6	103.3	600470	0.2	0	91.67%
maxTMaxSD	0.1	0.1	136	0.1	0	100.00%
minDom; MaxSD	118.5	839.5	3992698	95.5	0	33.33%
maxAvgVar; maxSD	143.5	707.0	2656630	68.4	0	46.67%
maxRegretVar; maxSD	0.1	0.1	0	0.1	0	100.00%
maxMinVar; maxSD	42.1	852.6	4340953	128.4	0	30.00%

Table .8 Average results for 40 TTPPV balanced instances (model 1)

TTPPV1 - Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	5.3	41.2	71186	0.9	252	97.00%
Brélaz; lexico	0.1	60.1	100661	0.1	0	95.00%
dom/ddeg; lexico	0.1	0.1	7	0.1	0	100.00%
IBS	8.1	8.1	4694	6.0	4320	100.00%
IBS+	12.1	15.0	12513	9.3	8098	99.75%
Ilog IBS+	12.8	12.8	9208	9.3	7949	100.00%
RSC+LA	226.0	226.0	0	134.9	0	100.00%
RSC2+LA	39.3	39.3	0	25.9	0	100.00%
Brélaz; maxSD	0.2	120.2	175869	0.3	0	90.00%
dom/ddeg; maxSD	19.8	108.3	151076	0.2	0	92.50%
IBS; maxSD	8.6	8.6	5317	6.3	4584	100.00%
IBS+; maxSD	8.8	8.8	5317	6.4	4584	100.00%
maxSD	0.4	0.4	6	0.3	0	100.00%
maxAggr(maxRelSD)	0.6	30.6	27248	0.4	0	97.50%
maxAggr(maxRelRatio)	0.7	0.7	88	0.3	15	100.00%
maxAggr(min)	0.4	0.4	68	0.3	0	100.00%
maxAggr(aAvg)	0.4	90.4	133863	0.6	0	92.50%
maxAggr(wSCAvg)	1.6	31.5	37215	0.4	0	97.50%
maxAggr(wAntiSCAvg)	0.4	30.4	38082	0.4	0	97.50%
maxAggr(wTAvg)	0.4	60.4	91481	0.5	0	95.00%
maxAggr(wAntiTAvg)	1.3	91.2	105966	0.7	0	92.50%
maxAggr(wDAvg)	0.4	0.4	3	0.3	0	100.00%
maxAggr(SCRemaining)	0.4	0.4	84	0.3	0	100.00%
minAggr(SCRemoved)	0.4	30.4	25742	0.4	0	97.50%
maxAggr(solProb)	0.4	120.4	126074	0.7	0	90.00%
maxAggr(avgSDDisc)	0.7	60.7	59963	0.5	0	95.00%
maxAggr(maxSDDisc)	0.8	60.8	59664	0.6	0	95.00%
minSCMaxSD	0.4	0.4	16	0.3	0	100.00%
minTMaxSD	0.3	0.3	16	0.2	0	100.00%
maxSCMaxSD	2.0	61.9	56642	0.5	0	95.00%
maxTMaxSD	0.4	30.4	37066	0.4	0	97.50%
minDom; MaxSD	0.2	30.2	41234	0.2	0	97.50%
maxAvgVar; maxSD	0.4	30.4	39395	0.4	0	97.50%
maxRegretVar; maxSD	0.4	0.4	6	0.3	0	100.00%
maxMinVar; maxSD	0.4	0.4	1	0.3	0	100.00%

Table .9 Average results for 40 TTPPV non-balanced instances (model 1)

TTPPV1 - Non-Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	20.6	770.2	1464778	455.3	767090	36.25%
Brélaz; lexico	0.2	960.0	2016570	186.1	239716	20.00%
dom/ddeg; lexico	2.3	870.7	1854841	111.5	152393	27.50%
IBS	58.2	514.9	561102	85.5	89597	60.00%
IBS+	49.6	512.6	705301	219.2	304771	59.50%
Ilog IBS+	25.9	554.2	853358	96.0	110422	55.00%
RSC+LA	232.8	643.9	608	340.5	0	57.50%
RSC2+LA	30.9	937.0	145440	472.6	0	22.50%
Brélaz; maxSD	0.2	1050.0	2011554	387.4	475368	12.50%
dom/ddeg; maxSD	0.5	1020.1	1962969	329.0	0	15.00%
IBS; maxSD	150.9	518.1	516610	114.2	110718	65.00%
IBS+; maxSD	154.4	520.4	505082	116.2	109453	65.00%
maxSD	58.5	1028.8	1423450	420.2	0	15.00%
maxAggr(maxRelSD)	52.5	654.9	662841	69.2	39824	47.50%
maxAggr(maxRelRatio)	43.2	621.6	623683	45.6	24515	50.00%
maxAggr(min)	143.7	1015.2	1270323	340.8	0	17.50%
maxAggr(aAvg)	0.4	870.1	1208806	129.1	0	27.50%
maxAggr(wSCAvg)	0.4	990.1	1400871	285.5	0	17.50%
maxAggr(wAntiSCAvg)	0.4	1080.1	1390025	529.4	0	10.00%
maxAggr(wTAvg)	0.5	900.1	1172868	164.4	0	25.00%
maxAggr(wAntiTAvg)	0.5	900.1	1198267	158.9	69530	25.00%
maxAggr(wDAvg)	6.3	931.4	1270167	213.8	125043	22.50%
maxAggr(SCRemaining)	54.3	770.4	770233	131.9	0	37.50%
minAggr(SCRemoved)	87.6	894.1	709382	148.8	0	27.50%
maxAggr(solProb)	0.6	990.1	884554	306.8	0	17.50%
maxAggr(avgSDDisc)	15.6	903.9	1076904	177.3	118206	25.00%
maxAggr(maxSDDisc)	16.2	904.1	1047311	178.4	115910	25.00%
minSCMaxSD	36.0	763.5	906973	91.2	67720	37.50%
minTMaxSD	36.8	763.8	893787	92.8	67115	37.50%
maxSCMaxSD	55.1	799.3	845149	102.6	0	35.00%
maxTMaxSD	11.6	843.5	932826	150.4	0	30.00%
minDom; MaxSD	42.5	997.5	1746597	307.9	0	17.50%
maxAvgVar; maxSD	0.8	930.2	1215478	189.3	0	22.50%
maxRegretVar; maxSD	92.1	839.9	906801	175.3	118381	32.50%
maxMinVar; maxSD	0.3	1050.1	1374437	410.1	0	12.50%

Table .10 Average results for 40 TTPPV balanced instances (model 2)

TTPPV2 - Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	2.5	11.5	34220	0.2	15	99.25%
Brélaz; lexico	0.1	0.1	9	0.1	0	100.00%
dom/ddeg; lexico	0.1	0.1	27	0.1	0	100.00%
IBS	8.0	8.0	5189	6.0	4488	100.00%
IBS+	10.9	10.9	8287	8.4	7583	100.00%
Ilog IBS+	10.8	10.8	8250	8.4	7555	100.00%
RSC+LA	207.6	207.6	0	131.9	0	100.00%
RSC2+LA	37.7	95.8	18155	32.9	0	95.00%
Brélaz; maxSD	0.1	180.1	436244	0.4	0	85.00%
dom/ddeg; maxSD	8.1	157.1	391736	0.5	0	87.50%
IBS; maxSD	8.7	38.4	22679	7.2	5630	97.50%
IBS+; maxSD	8.6	38.4	22303	7.2	5627	97.50%
maxSD	0.5	0.5	0	0.4	0	100.00%
maxAggr(maxRelSD)	0.6	0.6	0	0.4	0	100.00%
maxAggr(maxRelRatio)	0.5	0.5	1	0.4	0	100.00%
maxAggr(min)	28.7	145.8	227545	1.3	0	90.00%
maxAggr(aAvg)	0.5	0.5	0	0.4	0	100.00%
maxAggr(wSCAvg)	0.5	30.5	39926	0.5	0	97.50%
maxAggr(wAntiSCAvg)	0.5	0.5	2	0.4	0	100.00%
maxAggr(wTAvg)	0.6	0.6	0	0.4	0	100.00%
maxAggr(wAntiTAvg)	0.5	30.5	36930	0.5	0	97.50%
maxAggr(wDAvg)	0.6	0.6	6	0.4	0	100.00%
maxAggr(SCRemaining)	0.5	0.5	13	0.4	0	100.00%
minAggr(SCRemoved)	0.6	0.6	2	0.5	0	100.00%
maxAggr(solProb)	1.1	1.1	732	0.5	0	100.00%
maxAggr(avgSDDisc)	0.6	0.6	1	0.5	0	100.00%
maxAggr(maxSDDisc)	0.6	0.6	1	0.5	0	100.00%
minSCMaxSD	2.4	2.4	3782	0.4	0	100.00%
minTMaxSD	2.3	2.3	3782	0.4	0	100.00%
maxSCMaxSD	4.6	4.6	6770	0.4	0	100.00%
maxTMaxSD	0.5	0.5	57	0.4	0	100.00%
minDom; MaxSD	0.2	0.2	1	0.2	0	100.00%
maxAvgVar; maxSD	0.6	0.6	0	0.4	0	100.00%
maxRegretVar; maxSD	0.6	0.6	1	0.4	0	100.00%
maxMinVar; maxSD	0.5	150.5	191573	1.1	0	87.50%

Table .11 Average results for 40 TTPPV non-balanced instances (model 2)

TTPPV2 - Non-Balanced instances						
	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	15.7	768.8	2352684	376.2	952225	36.25%
Brélaz; lexico	4.1	901.0	2753370	128.6	164810	25.00%
dom/ddeg; lexico	4.7	901.2	2773954	135.6	195813	25.00%
IBS	67.1	775.2	1209574	209.0	271429	37.50%
IBS+	62.7	515.2	921688	307.5	561356	59.25%
llog IBS+	64.0	632.0	1056281	108.8	139389	50.00%
RSC+LA	205.5	553.6	697	284.5	0	65.00%
RSC2+LA	43.8	911.0	203565	486.5	0	25.00%
Brélaz; maxSD	0.1	1170.0	2671953	936.8	1854798	2.50%
dom/ddeg; maxSD	0.1	1170.0	2685464	932.6	1870735	2.50%
IBS; maxSD	209.4	952.4	916441	642.6	613131	25.00%
IBS+; maxSD	207.5	951.9	911345	642.7	611726	25.00%
maxSD	0.6	30.5	2855	0.5	0	97.50%
maxAggr(maxRelSD)	0.6	30.5	4112	0.5	0	97.50%
maxAggr(maxRelRatio)	0.5	30.5	4168	0.5	0	97.50%
maxAggr(min)	0.3	1170.1	1225404	977.6	0	2.50%
maxAggr(aAvg)	0.8	30.8	2914	0.5	0	97.50%
maxAggr(wSCAvg)	0.6	30.6	32713	0.5	0	97.50%
maxAggr(wAntiSCAvg)	0.6	90.5	8844	0.8	0	92.50%
maxAggr(wTAvg)	0.6	30.6	2735	0.5	0	97.50%
maxAggr(wAntiTAvg)	14.4	251.5	136553	2.8	0	80.00%
maxAggr(wDAvg)	0.5	330.4	281893	3.7	0	72.50%
maxAggr(SCRemaining)	69.3	747.7	875715	79.4	0	40.00%
minAggr(SCRemoved)	0.6	30.6	2354	0.6	0	97.50%
maxAggr(solProb)	99.4	1007.4	1172267	368.0	230716	17.50%
maxAggr(avgSDDisc)	7.5	7.5	682	0.5	0	100.00%
maxAggr(maxSDDisc)	7.4	7.4	682	0.5	0	100.00%
minSCMaxSD	61.1	829.9	1101319	153.3	0	32.50%
minTMaxSD	60.5	829.7	1115397	151.0	0	32.50%
maxSCMaxSD	32.2	849.7	1139561	174.0	113346	30.00%
maxTMaxSD	4.0	721.6	882025	59.9	29970	40.00%
minDom; MaxSD	34.9	297.0	279724	1.8	0	77.50%
maxAvgVar; maxSD	0.6	60.6	16937	0.6	0	95.00%
maxRegretVar; maxSD	26.6	789.3	703490	94.4	0	35.00%
maxMinVar; maxSD	N/A	1200.0	1232408	1200.0	982541	0.00%

Table .12 Aggregated average results over the eight problem domains

	a.T(S)	a.T	a.Bcks	g.T	g.Bcks	%sol
rndMinSizeRndVal	123.2	408.7	708994	347.9	241614	71.80%
Brélaz; lexico	94.0	447.0	690109	366.7	286357	67.55%
dom/ddeg; lexico	102.0	453.8	687008	370.7	290280	67.55%
IBS	70.5	352.7	237088	120.7	38937	74.90%
Ilog IBS+	197.0	367.9	315401	132.7	63062	82.59%
IBS+	252.4	423.6	422921	145.7	82068	77.70%
RSC+LA	313.5	536.2	4033	196.2	141	69.38%
RSC2+LA	187.9	536.0	49248	276.7	4788	64.48%
Brélaz; maxSD	104.1	458.6	1226556	337.3	230290	67.95%
dom/ddeg; maxSD	103.4	454.5	1235699	335.6	231490	68.21%
IBS; maxSD	132.9	307.8	162754	114.5	28293	84.27%
IBS+; maxSD	134.6	309.5	162263	115.1	26219	84.27%
maxSD	48.6	84.8	75450	31.7	5390	96.95%
maxAggr(maxRelSD)	57.8	115.9	48810	18.8	1168	94.86%
maxAggr(maxRelRatio)	189.2	316.1	204829	55.6	9014	79.66%
maxAggr(min)	123.8	468.5	727680	388.8	179689	68.91%
maxAggr(aAvg)	64.6	154.3	236063	81.1	20618	92.24%
maxAggr(wSCAvg)	62.0	222.2	64762	71.7	11980	86.24%
maxAggr(wAntiSCAvg)	66.9	217.6	496525	106.4	37555	86.68%
maxAggr(wTAvg)	89.8	293.7	559890	186.9	46665	81.47%
maxAggr(wAntiTAvg)	49.5	94.3	82113	31.1	9467	96.16%
maxAggr(wDAvg)	59.2	175.2	62041	43.9	6126	90.09%
maxAggr(SCRemaining)	64.9	523.3	333598	363.6	112977	60.55%
minAggr(SCRemoved)	124.2	419.6	501480	243.7	47550	72.68%
maxAggr(solProb)	94.3	405.6	246857	252.2	80705	72.23%
maxAggr(avgSDDisc)	89.5	189.3	95446	36.4	3300	89.81%
maxAggr(maxSDDisc)	94.7	215.5	107863	41.4	4200	88.06%
minSCMaxSD	110.5	337.6	310254	93.9	41573	77.82%
minTMaxSD	112.2	337.9	307022	92.8	41598	77.82%
maxSCMaxSD	98.4	503.2	437484	409.6	181646	63.52%
maxTMaxSD	124.9	456.5	313723	152.8	65638	68.83%
minDom; MaxSD	61.4	201.1	554933	107.5	42106	87.47%
maxAvgVar; maxSD	69.5	206.2	373354	106.7	28085	87.57%
maxRegretVar; maxSD	81.5	217.7	103825	34.4	2967	88.02%
maxMinVar; maxSD	108.4	459.3	757674	321.5	121198	68.64%